Harmless Advice [*]

Daniel S. Dantas
Princeton University
ddantas@cs.princeton.edu

David Walker
Princeton University
dpw@cs.princeton.edu

Abstract

This paper defines an object-oriented language with *harmless* aspect-oriented advice. A piece of harmless advice is a computation that, like ordinary aspect-oriented advice, executes when control reaches a designated control-flow point. However, unlike ordinary advice, harmless advice is designed to obey a weak non-interference property. Harmless advice may change the termination behavior of computations and use I/O, but it does not otherwise influence the final result of the mainline code. The benefit of harmless advice is that it facilitates local reasoning about program behavior. More specifically, programmers may ignore harmless advice when reasoning about the partial correctness properties of their programs. In addition, programmers may add new pieces of harmless advice to pre-existing programs in typical "after-the-fact" aspect-oriented style without fear they will break important data invariants used by the mainline code.

In order to detect and enforce harmlessness, the paper defines a novel type and effect system related to information-flow type systems. The central technical result is that well-typed harmless advice does not interfere with the mainline computation. The paper also presents an implementation of the language and a case study using harmless advice to implement security policies.

1. Introduction

Aspect-oriented programming languages (AOPL) such as AspectJ [17] allow programmers to specify both *what* computation to perform as well as *when* to perform it. For example, AspectJ makes it easy to implement a profiler that records statistics concerning the number of calls to each method: The *what* in this case is the computation that does the recording and the *when* is the instant of time just prior to execution of each method body. In aspect-oriented terminology, the specification of *what* to do is called *advice* and the specification of *when* to do it is called a *point cut*. A collection of point cuts and advice organized to perform a coherent task is called an *aspect*.

Many within the AOP community adhere to the tenet that aspects are most effective when the code they advise is *oblivious* to their presence [12]. 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 extremely flexible profiling infrastructure. To adjust the places where profiling occurs, which might be scattered all 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 your program 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. Consequently, it plays a role similar to Clifton and Leavens' notion of *observer* [4]. Since harmless advice does not interfere with the mainline computation, it 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 that 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 under-

[*] This research was supported in part by ARDA Grant no. NBCHC030106, NSF grants CCR-0238328 and CCR-0208601, and an Alfred P. Sloan Fellowship. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of the NSF, ARDA or Sloan foundation and no official endorsement should be inferred.

lying 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 [22] and Flow Caml [24]. 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 secret information at too great a rate.

In our case, an appropriate balance point between useability and interference prevention is even more relaxed than in secure information-flow systems. 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 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 favourite (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.

- 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.
- Invariant checking and security. Harmless advice can check invariants at run-time, maintain access control tables, perform resource accounting, and terminate programs that disobey dynamic security policies.
- Program tracing and monitoring. Harmless advice can print out all sorts 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 post-mortem security audits.

We have also accumulated anecdotal evidence that indicates enough important applications appear to fall into this category to make it a useful abstraction. For example, IBM experimented with aspects in their middleware product line [5], 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 [14] 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 profiling, security, and monitoring mentioned above. Finally, we have done a case study in security, rewriting the security policies from Evans' thesis [10] in our system, and finding that all but one was harmless.

In the rest of this paper, we develop a theory of harmless advice following the same strategy as used in previous work by Walker, Zdancewic and Ligatti [27] (hereafter referred to as WZL) and inspired by the type-theoretic definition of Standard ML [13]. This strategy involves defining type-safe languages at two levels of abstraction: a high-level, user-friendly source language for programmers and a lower-level semantic intermediate language we call the *core calculus*. The core calculus consists of a collection of simple, orthogonal constructs amenable to formal analysis. The source language is defined by translation into the core calculus. The primary reason for this structure is that the source language features are quite complex and a direct analysis of a deep property such as noninterference would be incredibly hard and very error-prone. We simplify the situation by proving a collection of powerful properties of the core calculus. Then, we show that the translation from source into core is type-preserving and exploit properties of well-typed core calculus constructs to obtain properties of the source. Overall, this structure helps modularize and simplify an otherwise daunting task.

Section 2 defines the core calculus. The calculus contains primitive notions of point cuts, advice and a collection of static protection domains, arranged in a partial order. We define a novel type system to guarantee that code, including advice, in a low-protection domain cannot influence execution of code in a high-protection domain. Though the type system is directly inspired by information-flow type systems for security [22, 24], we take advantage of the syntactic separation between advice and code to simplify it. The key technical result for the core language is a proof that the core language satisfies a non-interference property. The proof adapts the technique used by Simonet and Pottier in their proof of non-interference of Flow Caml [24] to a our aspect-oriented language.

Section 3 develops a surface language that is more amenable to program- ming. In particular, the high-level language is oblivious and therefore "aspect-oriented" according to Filman and Friedman's definition, whereas the core language is not.[1] The source 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 the source and establishes its semantics through a translation from source into core. We prove that source language aspects are harmless by exploiting properties of the core. In addition, we present example code implemented in our system and explain a case study in security we have performed.

Section 4 discusses related work in more detail and Section 5 concludes.

2. Core Language

Our core language is a typed lambda calculus containing strings, booleans, tuples, references and simple objects. The two main features of interest in the language are labeled control-flow points and advice, both of which are slight variants of related constructs introduced by WZL.

[1] It is neither necessary nor the slightest bit desirable for the core language to be oblivious as the syntax of the core language does not limit or constrain programmers in any way. Programmers need only concern themselves with the surface language, which is oblivious.

Labels l, which are drawn from some countably infinite set, mark points in a computation at which advice may be triggered. For instance, execution of $l\,\texttt{[}e_1\texttt{]}\,;e_2$ 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. A marked point $l\,\texttt{[}e_1\texttt{]}$ has type unit and that no data are returned from the triggered advice. This stands in contrast to earlier work by WZL, in which labels marked control-flow points where data exchange could occur.

Harmless advice $\{pcd.x \to e\}$ is a computation that is triggered whenever execution reaches the control-flow point described by the pointcut designator pcd. 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.

Languages such as AspectJ often contain rich sublanguages for designating control-flow points. However, it is easier to study the fundamentals of labeled control-flow points and harmless advice in a setting with the simplist possible pcds. Consequently, we will start our investigation in a setting where pcds are simply sets of labels $\{l_1, \ldots, l_k\}$ and advice is written as $\{\{l_1, \ldots, l_k\}.x \to e\}$.

For simplicity, the core language contains a single construct $\Uparrow a$ to activate new advice a. When control reaches a label in the advice's point cut designator, the advice body will execute after any previously activated piece of advice. The following example shows how advice activation works (assuming that there is no other advice associated with label l in the environment). The code prints `3: hello world`.

$$
\begin{array}{l}
\Uparrow \{\{l\}.x \to \texttt{printint x; print " : hello "}\};\\
\Uparrow \{\{l\}.y \to \texttt{print "world"}\};\\
l\,\texttt{[3]}
\end{array}
$$

The expression $\texttt{new} : \tau$ 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.

$$
\begin{array}{l}
\texttt{let}\, pt = \texttt{new} : \texttt{int}\,\texttt{in}\\
\Uparrow \{\{pt\}.x \to \texttt{print "hello "}\};\\
\Uparrow \{\{pt\}.y \to \texttt{print "world"}\};\\
pt\,\texttt{[3]}
\end{array}
$$

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 $\mathrm{L} = (\mathit{Protections}, \leq)$ where $\mathit{Protections}$ is the set of possible protection domains and $p \leq q$ specifies that p should not interfere with q. Alternatively, one might say that data in q have higher integrity than data in p. In our examples, we often assume there are `high`, `med` and `low` protection levels with `low < med < high`.

Syntax 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 1.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{p \in \text{Protections} \qquad l \in \text{Labels} \qquad \texttt{s} \in \text{Strings}}\\
\tau & ::= & \texttt{unit} \mid \texttt{string} \mid \texttt{bool} \mid \tau_1 \times \ldots \times \tau_n\\
 & \mid & \tau \to_p \tau \mid [m_i{:}_{p_i}\tau_i]^{1..n}\\
 & \mid & \texttt{advice}_p \mid \tau\,\texttt{label}_p \mid \tau\,\texttt{ref}_p \mid \tau\,\texttt{pcd}_p\\
v & ::= & \texttt{()} \mid \texttt{s} \mid \texttt{true} \mid \texttt{false} \mid (\vec{v})\\
 & \mid & \lambda_p x : \tau.e \mid [m_i = \varsigma_p\, x_i.e_i]^{1..n} \mid \{v.x \to_p e\}\\
 & \mid & l \mid r \mid \{\vec{l}\}_p\\
e & ::= & v \mid x \mid e_1; e_2 \mid \texttt{print}\, e\\
 & \mid & \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3\\
 & \mid & (\vec{e}) \mid \texttt{split}\,(\vec{x}) = e\,\texttt{in}\,e\\
 & \mid & e\,e \mid e.m \mid \{e.x \to_p e\} \mid \Uparrow e\\
 & \mid & \texttt{new}_p : \tau \mid e\,\texttt{[}e\texttt{]}\\
 & \mid & \texttt{ref}_p\, e \mid \texttt{!}\,e \mid e := e\\
 & \mid & \{\vec{e}\}_p \mid e \cup_p e \mid p\texttt{<}e\texttt{>}
\end{array}
$$

Figure 1. Core Language Syntax

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. Objects are collections of methods, with each method taking a single parameter (self). Methods and object types are also annotated with protection domains. Advice values $\{v.x \to_p e\}$ 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 $\texttt{new}_p : \tau$ and $\texttt{ref}_p\ e$ 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.

2.2 Typing

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 2 and 3 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 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 context). 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.

Most of the value typing rules 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, point-cut designators are sets of labels. Unlike the other values, the rules for typing advice are fairly subtle. We will discuss these rules in a moment together with the rules for typing labeled control-flow points.

The first few expression typing rules (see Figure 3) are standard rules for type systems that track information flow. The rule for 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. 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 and method invocations require that the function or method in question be safe to run at the current protection level p.

The typing rules for references enforce the usual integrity constraint found in information-flow systems. When in protection domain p, we are allowed to dereference references in protection domain p' when p is less than or equal to p'. We are allowed to store to references in protection domain p' only if our current domain p is greater than or equal to p'.

The last rule in Figure 3 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 `unit`. One might wonder whether the following dual rule, which allows one to raise the protection level is sound in our system:

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

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 2.4, it is not. Fortunately, the rule does not appear useful in our application and we have omitted it.[2]

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 `int` $\texttt{ref}_{\texttt{high}}$ and l is a label that has been placed in high-protection code. If we allow the expression
$$\{l.x \rightarrow_{\texttt{high}} r := 3 + x\} \texttt{<<} \; e$$
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
$$\{l.x \rightarrow_{\texttt{high}} r := 3 + x\}$$
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.

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., $\texttt{advice}_{\texttt{high}}$), 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., `string` $\texttt{label}_{\texttt{high}}$) 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.

Five typing rules in the middle of Figure 3 give the well-formedness conditions for advice and labels. Notice that in the rule for typing advice introduction, 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. Notice also that when marking a control-flow point with a label, the protection level of the label is connected to the protection level of the expression at that point. Finally, given a high-protection piece of advice, this advice cannot be launched from low-protection code. The result of these constraints is that when in a low-protection zone, there is no way to cause execution of high-protection advice.

2.3 Operational Semantics

The definition of the operational semantics for our language largely follows earlier work by WZL. 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 judgment has the form $(S, A, p, e) \longmapsto (S', A', p, e')$ where S 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 $(S, A, p, e) \longmapsto_\beta (S', A', p, e')$. The syntax of stores and advice stores is given below.

$$\begin{array}{lcl} S & ::= & \cdot \mid S, r = e \mid S, l \\ A & ::= & \cdot \mid A, \{v.x \rightarrow_p e\} \end{array}$$

The definitions of these relations can be found in Figure 4. Notice that the rule for marked control-flow points depends upon an auxiliary function $\mathcal{A}[\![A]\!]_{l\,[v]} = e$. This function selects all advice in A that is triggered by the label l and combines their bodies to form the expression e. The advice composition function can be found in Figure 5. Finally, an abstract machine configuration (S, A, p, e) is well-typed if it satisfies the judgement $\vdash (S, A, p, e)$ `ok` specified in Figure 6.

[2] There may well be some strategy that allows us to add this rule together with the context-sensitive advice of Section 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.

$$\frac{}{\Gamma \vdash \texttt{()} : \texttt{unit}} \qquad \frac{}{\Gamma \vdash \texttt{s} : \texttt{string}}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}$$

$$\frac{(\Gamma \vdash v_i : \tau_i)^{1 \le i \le n}}{\Gamma \vdash (\vec{v}) : \tau_1 \times ... \times \tau_n} \qquad \frac{\Gamma, x : \tau; p \vdash e : \tau'}{\Gamma \vdash \lambda_p x : \tau.e : \tau \to_p \tau'}$$

$$\frac{((\Gamma, x : [m_i{:}_{p_i}\tau_i]^{1..n}); p_j \vdash e_j : \tau_j)^{(1 \le j \le n)}}{\Gamma \vdash [m_i = \varsigma_{p_i}\, x_i.e_i]^{1..n} : [m_i{:}_{p_i}\tau_i]^{1..n}}$$

$$\frac{\Gamma \vdash v : \tau\ \texttt{pcd}_p \quad \Gamma, x : \tau; p' \vdash e : \texttt{unit} \quad \vdash p' \le p}{\Gamma \vdash \{v.x \to_{p'} e\} : \texttt{advice}_{p'}}$$

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

$$\frac{(\Gamma \vdash v_i : \tau\ \texttt{label}_{p_i})^{(1 \le i \le n)} \quad (\vdash p \le p_i)^{(1 \le i \le n)}}{\Gamma \vdash \{\vec{l}\}_p : \tau\ \texttt{pcd}_p}$$

Figure 2. Core Value Typing

2.4 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. In this section, we describe a fully general facility for analysis of information on the current call stack. Our new mechanism is inspired by earlier work by WZL, but is more general and fits better with the functional programming paradigm. Figure 7 outlines the syntactic extensions to our calculus.

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 $\texttt{store}\ e_1\,\texttt{[}e_2\texttt{]}\ \texttt{in}\ e_3$, 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 $l\,\texttt{[}v_2\texttt{]}$ on the stack, evaluates e_3 to a value v_3 and finally removes $l\,\texttt{[}v_2\texttt{]}$ from the stack and returns v_3. The programmer may examine a stack data structure using the $\texttt{case}\ e\ \texttt{of}\ (pat \Rightarrow e \mid \_ \Rightarrow e)$ 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 (e.g., $\cdot$), patterns that match a stack starting with any label in a particular set (e.g., $\{\vec{l}\}_p\,\texttt{[}x\texttt{]}\ ::\ pat$) 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 (e.g., $\_ :: pat$), and patterns involving stack variables (e.g., x).

The typing rules for these extensions appear in Figure 8. There are three sets of rules in this figure. The first two 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 9. Again, there are three sets of rules. The first defines a new set of top-level evaluation rules, and the second 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<F> contexts. Here, we use the notation $st@X$

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

$$\frac{\Gamma; p \vdash e_1 : \texttt{unit} \quad \Gamma; p \vdash e_2 : \tau}{\Gamma; p \vdash e_1; e_2 : \tau}$$

$$\frac{\Gamma; p \vdash e : \texttt{string}}{\Gamma; p \vdash \texttt{print}\ e : \texttt{unit}}$$

$$\frac{\Gamma; p \vdash e_1 : \texttt{bool} \quad \Gamma; p \vdash e_2 : \tau \quad \Gamma; p \vdash e_3 : \tau}{\Gamma; p \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

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

$$\frac{\Gamma; p \vdash e_1 : \tau_1 \times ... \times \tau_n \quad \Gamma, (\vec{x} : \vec{t}); p \vdash e_2 : \tau}{\Gamma; p \vdash \texttt{split}\,(\vec{x}) = e_1\ \texttt{in}\ e_2 : \tau}$$

$$\frac{\Gamma; p \vdash e_1 : \tau_1 \to_p \tau_2 \quad \Gamma; p \vdash e_2 : \tau_1}{\Gamma; p \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma; p \vdash e : [m_i{:}_{p_i}\tau_i]^{1..n} \quad 1 \le j \le n \quad p = p_j}{\Gamma; p \vdash e.m_j : \tau_j}$$

$$\frac{\Gamma; p \vdash e_1 : \tau\ \texttt{pcd}_{p'} \quad \Gamma, x : \tau; p'' \vdash e_2 : \texttt{unit} \quad \vdash p'' \le p'}{\Gamma; p \vdash \{e_1.x \to_{p''} e_2\} : \texttt{advice}_{p''}}$$

$$\frac{\Gamma; p \vdash e : \texttt{advice}_{p'} \quad \vdash p' \le p}{\Gamma; p \vdash \Uparrow e : \texttt{unit}}$$

$$\frac{\vdash p' \le p}{\Gamma; p \vdash \texttt{new}_{p'} : \tau : \tau\ \texttt{label}_{p'}}$$

$$\frac{\Gamma; p \vdash e_1 : \tau\ \texttt{label}_p \quad \Gamma; p \vdash e_2 : \tau}{\Gamma; p \vdash e_1\,\texttt{[}e_2\texttt{]} : \texttt{unit}}$$

$$\frac{\Gamma; p \vdash e : \tau \quad \vdash p' \le p}{\Gamma; p \vdash \texttt{ref}_{p'}\ e : \tau\ \texttt{ref}_{p'}} \qquad \frac{\Gamma; p \vdash e : \tau\ \texttt{ref}_{p'} \quad \vdash p \le p'}{\Gamma; p \vdash !e : \tau}$$

$$\frac{\Gamma; p \vdash e_1 : \tau\ \texttt{ref}_{p'} \quad \Gamma; p \vdash e_2 : \tau \quad \vdash p' \le p}{\Gamma; p \vdash e_1 := e_2 : \tau}$$

$$\frac{(\Gamma; p \vdash e_i : \tau\ \texttt{label}_{p_i})^{(1 \le i \le n)} \quad (\vdash p' \le p_i)^{(1 \le i \le n)}}{\Gamma; p \vdash \{\vec{e}\}_{p'} : \tau\ \texttt{pcd}_{p'}}$$

$$\frac{\begin{array}{ll}\Gamma; p \vdash e_1 : \tau\ \texttt{pcd}_{p''} & \vdash p' \le p'' \\ \Gamma; p \vdash e_2 : \tau\ \texttt{pcd}_{p'''} & \vdash p' \le p'''\end{array}}{\Gamma; p \vdash e_1 \cup_{p'} e_2 : \tau\ \texttt{pcd}_{p'}}$$

$$\frac{\Gamma; p' \vdash e : \texttt{unit} \quad \vdash p' \le p}{\Gamma; p \vdash p'\texttt{<}e\texttt{>} : \texttt{unit}}$$

Figure 3. Core Expression Typing

$$\frac{(S, A, p, e) \longmapsto_{\beta} (S', A', p, e')}{(S, A, p, e) \longmapsto (S', A', p, e')}$$

$$\frac{(S, A, p, e) \longmapsto (S', A', p, e)}{(S, A, p, E[e]) \longmapsto (S', A', p, E[e'])}$$

$$\frac{(S, A, p', e) \longmapsto (S', A', p', e')}{(S, A, p, p'\texttt{<}e\texttt{>}) \longmapsto (S', A', p, p'\texttt{<}e'\texttt{>})}$$

$$(S, A, p, (); e) \longmapsto_{\beta} (S, A, p, e)$$

$$(S, A, p, \texttt{print}\ s) \longmapsto_{\beta} (S, A, p, ())$$

$$(S, A, p, \texttt{if}\ \mathit{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2) \longmapsto_{\beta} (S, A, p, e_1)$$

$$(S, A, p, \texttt{if}\ \mathit{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2) \longmapsto_{\beta} (S, A, p, e_2)$$

$$(S, A, p, \texttt{split}\ (\vec{x}) = (\vec{v})\ \texttt{in}\ e) \longmapsto_{\beta} (S, A, p, e\{\vec{v}/\vec{x}\})$$

$$(S, A, p, \lambda_p x : t.e\ v) \longmapsto_{\beta} (S, A, p, e\{v/x\})$$

$$\begin{array}{l}(S, A, p, [m_i = \varsigma_{p_i}\ x_i.e_i]^{1..n}.m_j) \longmapsto_{\beta} \\ \quad (S, A, p, e_j\{[m_i = \varsigma_{p_i}\ x_i.e_i]^{1..n}/x_j\})\end{array}$$

$$(S, A, p, \Uparrow \{v.x \rightarrow_{p'} e_1\}) \longmapsto_{\beta} (S, (A, \{v.x \rightarrow_{p'} e_1\}), p, ())$$

$$(l \notin S) \quad (S, A, p, \texttt{new}_{p'} : \tau) \longmapsto_{\beta} ((S, l), A, p, l)$$

$$\frac{l \in S \quad \mathcal{A}[\![A]\!]_{l\texttt{[}v\texttt{]}} = e}{(S, A, p, l\texttt{[}v\texttt{]}) \longmapsto_{\beta} (S, A, p, e)}$$

$$(r \notin S) \quad (S, A, p, \texttt{ref}_{p'}\ v) \longmapsto_{\beta} ((S, r = v), A, p, r)$$

$$(S, A, p, !\ r) \longmapsto_{\beta} (S, A, p, S(r))$$

$$(S, A, p, r := v) \longmapsto_{\beta} ((S, r = v), A, p, v)$$

$$(S, A, p, \{\vec{l_1}\}_{p'} \cup_{p''} \{\vec{l_2}\}_{p'''}) \longmapsto_{\beta} (S, A, p, \{\vec{l_1}\ \vec{l_2}\}_{p''})$$

$$(S, A, p, p'\texttt{<()>}) \longmapsto_{\beta} (S, A, p, ())$$

Figure 4. Core Operational Semantics

$$\frac{}{\mathcal{A}[\![\cdot]\!]_{l\texttt{[}v\texttt{]}} = ()}$$

$$\frac{l\texttt{[}v\texttt{]} \models v' \quad \mathcal{A}[\![A]\!]_{l\texttt{[}v\texttt{]}} = e}{\mathcal{A}[\![\{v'.x \rightarrow_p e'\}, A]\!]_{l\texttt{[}v\texttt{]}} = p\texttt{<}e'\{v/x\}\texttt{>}; e}$$

$$\frac{l\texttt{[}v\texttt{]} \not\models v' \quad \mathcal{A}[\![A]\!]_{l\texttt{[}v\texttt{]}} = e}{\mathcal{A}[\![\{v'.x \rightarrow_p e'\}, A]\!]_{C} = e}$$

$$\frac{l \in \{\vec{l}\}_p}{l\texttt{[}v\texttt{]} \models \{\vec{l}\}_p}$$

Figure 5. Core Aspect Composition

$$\frac{\begin{array}{c} \mathit{dom}(S) = \mathit{dom}(\Gamma) \\ \forall r \in \mathit{dom}(S).\ \Gamma(r) = \tau\ \texttt{ref}_p \ \ \Gamma \vdash S(r) : \tau\ \texttt{for some}\ p, \tau \\ \forall l \in \mathit{dom}(S).\ \Gamma(r) = \tau\ \texttt{ref}_p\ \texttt{for some}\ p, \tau \end{array}}{\vdash S : \Gamma}$$

$$\frac{}{\Gamma \vdash \cdot\ \texttt{ok}} \qquad \frac{\Gamma \vdash a : \texttt{advice}_p\ \texttt{for some}\ p \quad \Gamma \vdash A\ \texttt{ok}}{\Gamma \vdash A, a\ \texttt{ok}}$$

$$\frac{\vdash S : \Gamma \quad \Gamma \vdash A\ \texttt{ok} \quad \Gamma; p \vdash e : \tau\ \texttt{for some}\ \tau}{\vdash (S, A, p, e)\ \texttt{ok}}$$

Figure 6. Core Abstract Machine Judgement

$$\begin{array}{rcl}
\tau & ::= & \ldots \mid \texttt{stack} \\[1ex]
v & ::= & \ldots \mid \ \cdot \ \mid l\texttt{[}v\texttt{]} :: v \\[1ex]
e & ::= & \ldots \mid \texttt{stack()} \mid \texttt{store}\ e\texttt{[}e\texttt{]}\ \texttt{in}\ e \\
 & \mid & \texttt{case}\ e\ \texttt{of}\ \texttt{(}\mathit{pat} \Rightarrow e \mid \_ \Rightarrow e\texttt{)} \\[1ex]
\mathit{pat} & ::= & \texttt{nil} \mid e\texttt{[}x\texttt{]} :: \mathit{pat} \\
 & \mid & \_ :: \mathit{pat} \mid x \\[1ex]
\mathit{vpat} & ::= & \texttt{nil} \mid \{\vec{l}\}_p\texttt{[}x\texttt{]} :: \mathit{vpat} \\
 & \mid & \_ :: \mathit{vpat} \mid x \\[1ex]
E & ::= & \ldots \mid \texttt{store}\ E\texttt{[}e\texttt{]}\ \texttt{in}\ e \mid \texttt{store}\ l\texttt{[}E\texttt{]}\ \texttt{in}\ e \\
 & \mid & \texttt{store}\ l\texttt{[}v\texttt{]}\ \texttt{in}\ E \\
 & \mid & \texttt{case}\ E\ \texttt{of}\ \texttt{(}\mathit{pat} \Rightarrow e \mid \_ \Rightarrow e\texttt{)} \\
 & \mid & \texttt{case}\ v\ \texttt{of}\ \texttt{(}\mathit{Epat} \Rightarrow e \mid \_ \Rightarrow e\texttt{)} \\[1ex]
\mathit{Epat} & ::= & \ldots \mid E\texttt{[}x\texttt{]} :: \mathit{pat} \mid \{\vec{l}\}_p\texttt{[}x\texttt{]} :: \mathit{Epat} \\
 & \mid & \_ :: \mathit{Epat} \\[1ex]
F & ::= & \ldots \mid \texttt{[]} \mid E[F] \mid p < F >
\end{array}$$

Figure 7. Context Sensitive Advice Syntax

to append the object X to the bottom of the stack st. The last set of rules conclude in judgments with the form $st \models \mathit{vpat} \Rightarrow \mathit{sub}$. 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 possesses. 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<E>, which lowers the protection level. Consequently, any use of the `stack()` command is done in the context that looks like p_1<$E_1[p_2$<$E_2[p_3$<E_3>]>]> 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 `scase` expressions, the opposite is not possible. We are safe.

$\boxed{\Gamma \vdash v : \tau}$

$$\frac{}{\Gamma \vdash \cdot : \texttt{stack}}$$

$$\frac{\Gamma \vdash l : \tau\ \texttt{label}_p \quad \Gamma \vdash v_1 : \tau \quad \Gamma \vdash v_2 : \texttt{stack}}{\Gamma \vdash l\,\texttt{[}v_1\texttt{]} :: v_2 : \texttt{stack}}$$

$\boxed{\Gamma; p \vdash e : \tau}$

$$\frac{}{\Gamma; p \vdash \texttt{stack()} : \texttt{stack}}$$

$$\frac{\Gamma; p \vdash e_1 : \tau'\ \texttt{label}_{p'} \quad \Gamma; p \vdash e_2 : \tau' \quad \Gamma; p \vdash e_3 : \tau}{\Gamma; p \vdash \texttt{store}\ e_1\,[e_2]\ \texttt{in}\ e_3 : \tau}$$

$$\frac{\begin{array}{c}\Gamma; p \vdash e_1 : \texttt{stack} \\ \Gamma; p \vdash pat \Rightarrow \Gamma' \quad \Gamma, \Gamma'; p \vdash e_2 : \tau \quad \Gamma; p \vdash e_3 : \tau\end{array}}{\Gamma; p \vdash \texttt{case}\ e_1\ \texttt{of}\ (pat \Rightarrow e_2 \mid \_ \Rightarrow e_3) : \tau}$$

$\boxed{\Gamma; p \vdash pat \Rightarrow \Gamma}$

$$\frac{}{\Gamma; p \vdash \texttt{nil} \Rightarrow \cdot} \qquad \frac{\Gamma; p \vdash e : \tau\ \texttt{pcd}_{p'} \quad \Gamma; p \vdash pat \Rightarrow \Gamma'}{\Gamma; p \vdash e\,\texttt{[}x\texttt{]} :: pat \Rightarrow \Gamma', x : \tau}$$

$$\frac{\Gamma; p \vdash pat \Rightarrow \Gamma'}{\Gamma; p \vdash \_ :: pat \Rightarrow \Gamma'} \qquad \frac{}{\Gamma; p \vdash x \Rightarrow \cdot, x : \texttt{stack}}$$

Figure 8. Advanced Point-cut Designator Typing

$\boxed{(S, A, p, e) \longmapsto_{top} (S', A', p, e')}$

$$\frac{(S, A, p, e) \longmapsto (S', A', p, e')}{(S, A, p, e) \longmapsto_{top} (S', A', p, e')}$$

$$(S, A, p, F[\texttt{stack()}]) \longmapsto_{top} (S, A, p, F[\mathcal{S}(F)])$$

where :

$$\begin{array}{rcl} \mathcal{S}(\texttt{[]}) & = & \cdot \\ \mathcal{S}(\texttt{store}\ l\,\texttt{[}v\texttt{]}\ \texttt{in}\ F) & = & \mathcal{S}(F)\,@\,(l\,\texttt{[}v\texttt{]}) \\ \mathcal{S}(p < F >) & = & \mathcal{S}(F) \\ \mathcal{S}(E[F]) & = & \mathcal{S}(F) \\ & & \text{when } E \neq \texttt{store}\ l\,\texttt{[}v\texttt{]}\ \texttt{in}\ F \end{array}$$

$\boxed{(S, A, p, e) \longmapsto_{\beta} (S, A, p, e)}$

$$\frac{}{(S, A, p, \texttt{store}\ v_1\,\texttt{[}\tau\texttt{]}\ \texttt{in}\ v_2) \longmapsto_{\beta} (S, A, p, v_2)}$$

$$\frac{v \models vpat \Rightarrow sub}{\begin{array}{c}(S, A, p, \texttt{case}\ v\ \texttt{of}\ (vpat \Rightarrow e_1 \mid \_ \Rightarrow e_2)) \longmapsto_{\beta} \\ (S, A, p, sub(e_1))\end{array}}$$

$$\frac{v \not\models vpat \Rightarrow sub}{\begin{array}{c}(S, A, p, \texttt{case}\ v\ \texttt{of}\ (vpat \Rightarrow e_1 \mid \_ \Rightarrow e_2)) \longmapsto_{\beta} \\ (S, A, p, e_2)\end{array}}$$

$\boxed{v \models vpat \Rightarrow sub}$

$$\frac{}{\cdot \models \texttt{nil} \Rightarrow \cdot}$$

$$\frac{l \in \{\vec{l}\}_p \quad v_2 \models vpat \Rightarrow sub}{l\,\texttt{[}v_1\texttt{]} :: v_2 \models \{\vec{l}\}_p\,\texttt{[}x\texttt{]} :: vpat \Rightarrow sub, \{v_1/x\}}$$

$$\frac{v_2 \models vpat \Rightarrow sub}{l\,\texttt{[}v_1\texttt{]} :: v_2 \models \_ :: vpat \Rightarrow sub}$$

$$\frac{}{v \models x \Rightarrow \{v/x\}}$$

Figure 9. Advanced Point-cut Designator Evaluation

2.5 Core Language Meta-theory

To prove noninterference, we use the technique developed by Simonet and Pottier [24]. 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 10):

- Define a new language `Core2` that simulates execution of two original (henceforth referred to as `Core1`) programs.
- Show `Core2` is a correct simulation of `Core1` programs via Soundness and Completeness theorems.
- Prove `Core2` is a safe language and preserves the non-interference invariants via the standard Progress and Preservation theorems.
- Put the theorems above together to prove the noninterference result for `Core1`.

2.5.1 Defining `Core2`

We begin by defining a new language (`Core2`) that simulates execution of two of our original programs. The main syntactic difference between `Core1` expressions and `Core2` expressions is the brackets expression, p<$e_1|e_2$>. Here p is a low-protection label and the e_i are `Core1` expressions. These brackets expressions encapsulate all differences between the two `Core1` expressions that are being simulated. For instance, the `Core2` expression

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

represents the two `Core1` programs

```
p<print ‘‘hi’’>;x+3
```

```
p<print ‘‘bi’’>;x+3
```

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

$$\frac{\begin{array}{cc}\Gamma; p' \vdash_2 e_1 : \texttt{unit} & \Gamma; p' \vdash_2 e_2 : \texttt{unit} \\ p \in H \quad p' \in L & \vdash_2 p' \leq p\end{array}}{\Gamma; p \vdash_2 p'\texttt{<}e_1\,|\,e_2\texttt{>} : \texttt{unit}}$$

To express the operational semantics of `Core2` we add need to add similar bracket constructs for the contents of the reference/label

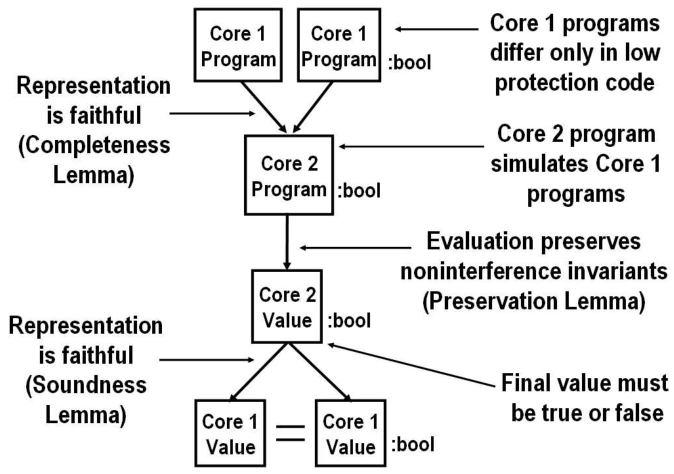
Core 1 Program
Core 1 Program
:bool
Core 1 programs differ only in low protection code
Representation is faithful (Completeness Lemma)
Core 2 Program
:bool
Core 2 program simulates Core 1 programs
Evaluation preserves noninterference invariants (Preservation Lemma)
Core 2 Value
:bool
Representation is faithful (Soundness Lemma)
Final value must be true or false
Core 1 Value
=
Core 1 Value
:bool

Figure 10. Noninterference Proof Diagram

store S and the aspect store A.

$$\begin{array}{rcl} v^2 & ::= & v \mid \texttt{<}v\,|\,v\texttt{>} \mid \texttt{<}v\,|\,void\texttt{>} \mid \texttt{<}void\,|\,v\texttt{>} \\ \tau^2 & ::= & \tau \mid \texttt{<}\tau\,|\,void\texttt{>} \mid \texttt{<}void\,|\,\tau\texttt{>} \\ a^2 & ::= & v \mid \texttt{<}v\,|\,void\texttt{>} \mid \texttt{<}void\,|\,v\texttt{>} \\ S & ::= & \cdot \mid S, r = v^2 \mid S, l \to \tau^2 \\ A & ::= & \cdot \mid A, a^2 \end{array}$$

The *void* marker indicates that the appropriate element is not present in the program. For example, if advice a is activated in only the left instance but not the right instance of simultaneously executing `Core1` programs, the aspect store of the `Core2` program that simulates them will contain $\texttt{<}a\,|\,void\texttt{>}$.

To relate `Core1` to `Core2`, we define the projection function $|\ |_i$ where i $\in$ 1,2. $|p\texttt{<}e_1|e_2\texttt{>}|_i$ is $p\texttt{<}e_i\texttt{>}$ and $|\ |_i$ is a homomorphism on all other expressions. Since $p\texttt{<}e_1|e_2\texttt{>}$ in `Core2` simulates the simultaneous execution of two low-protection original `Core1` expressions, the projection function extracts one of these two executions.

The `Core2` machine state (S, A, p, e) symbolizes the current state of the two simultaneously executing `Core1` programs where the i-th projection is the state of the i-th `Core1` program:

$$|(S, A, p, e)|_i = (|S|_i, |A|_i, p, |e|_i)$$

The projection function for the reference/label store and the aspect store is similar to the one for expressions.

The definition of the complete operational semantics of `Core2` is not too difficult, but it does involve a substantial amount of notation and due to space considerations, we omit the details. Intuitively, the main ideas are as follows:

- Ordinary `Core1` expressions embedded within `Core2` expressions operate as `Core1` expressions normally do.
- To evaluate inside the brackets expression $p\texttt{<}e_1|e_2\texttt{>}$, the semantics nondeterministically chooses one of e_1 or e_2 to execute. Operations on references or aspects 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.
- To evaluate the expression $p\texttt{<()|()>}$, 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 v_1 and v_2 are not both `()`, the expression $p\texttt{<}v_1|v_2\texttt{>}$ 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 `Core2` expressions has the same shape as the judgment form for `Core1` expressions. The index 2 on the arrow distinguishes the two judgements:

$$(S, A, p, e) \longmapsto^*_{2,top} (S', A', p, e')$$

2.5.2 Relating `Core1` and `Core2`

Once `Core2` has been defined, it is necessary to show that it accurately simulates two `Core1` programs. Two theorems, one concerning the *soundness* of `Core2` execution relative to `Core1` and the other concerning the *completeness* of `Core2` relative to `Core1` help to establish the proper correspondence.

The soundness theorem states that if a `Core2` expression takes a step, then the two corresponding `Core1` programs (the projections of the `Core2` 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.

THEOREM 2.1 (Soundness). *For $i \in \{1, 2\}$,*
if $(S, A, p, e) \longmapsto^*_{2,top} (S', A', p, e')$
then $|(S, A, p, e)|_i \longmapsto^*_{top} |(S', A', p, e')|_i$

The completeness theorem states that if two `Core1` programs step to values, then the representation in `Core2` that simulates them simultaneously must step to a value. The completeness theorem requires an auxiliary lemma stating that a `Core2` program is only stuck when one of its corresponding `Core1` programs are stuck.

THEOREM 2.2 (Completeness). *Assume $|(S, A, p, e)|_i \longmapsto^*_{top} (S'_i, A'_i, p, v_i)$ for all $i \in 1, 2$ then there exists (S', A', p, v) such that $(S, A, p, e) \longmapsto^*_{2,top} (S', A', p, v)$*

2.5.3 Safety of `Core2`

To continue we prove that the type system of `Core2` 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 `Core2` configuration (S, A, p, e) is well-typed if it satisfies the judgement $\vdash_2 (S, A, p, e)$ `ok` (omitted due to space constraints). 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 notice here 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.

We now state the standard Progress and Preservation lemmas.

THEOREM 2.3 (Progress). *If $\vdash_2 (S, A, p, e)$ `ok` then either e is a value, or there exists (S', A', p, e') such that $(S, A, p, e) \longmapsto_{2,top} (S', A', p, e')$.*

THEOREM 2.4 (Preservation). *If $\vdash_2 (S, A, p, e)$ `ok` and $(S, A, p, e) \longmapsto_{2,top} (S', A', p, e')$ then $\vdash_2 (S', A', p, e')$ `ok`.*

2.5.4 Putting it all together: Noninterference

Finally, for the noninterference proof, we assume a high-protection `Core1` expression e steps to a value. We add a low-protection expression $low\texttt{<}e'\texttt{>}$ where $low \in L$ to e 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

`Core2` expression low`<`$()|e'$`>`$;e$ where the right projection steps to e alone and the left projection is the low-protection code e' composed with 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 2.5 (Noninterference). *If $high \in H$ and $low \in L$ and $\vdash low \leq high$ and e is a core language expression where $\cdot; high \vdash e : \texttt{bool}$ and $\cdot; low \vdash e' : \texttt{unit}$ and $(\cdot, \cdot, high, low\texttt{<}e'\texttt{>}; e) \longmapsto^{*}_{top} (S_1, A_1, high, v_1)$ and $(\cdot, \cdot, high, low\texttt{<()>}; e) \longmapsto^{*}_{top} (S_2, A_2, high, v_2)$ then $v_1 = v_2$.*

3. Source Language

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 core calculus 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 type-correct core language 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 a user-friendly aspect-oriented source language with harmless advice. We have implemented the language in SML and explored the extent to which we can use harmless aspects to implement dynamic security policies.

3.1 Syntax

Figure 11 presents the formal syntax of the source language. The types of the source language objects are a restricted form of the internal language types. In particular, source language object types are the composition of a core language object and function type. Also, since programmers in the source language do not explicitly manipulate labels, there are no label types in the source language.

Most of the source language expressions and values mimic the core language 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 the source language is not executed directly. Also, for convenience, we allow a local let declaration in expressions, which programmers can use to allocate values with basic type, references or objects. Note that we use the meta-variable o to stand for program variables bound to objects. We use the meta-variable x to stand for any kind of program variable.

The source language case 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 method, we will automatically allocate the following items on the stack: a label corresponding to the method, a tuple containing a pointer to self, a pointer to the method argument, and a string corresponding to the name of the method that was called. Consequently, the patterns that match stack frames have the form $\{o.m_i\}^{1..n}[x, y, n]$, where $\{o.m_i\}^{1..n}$ is checked against the label, and x, y, and n are bound to self, the argument and the string

$$
\begin{array}{rcl}
\tau & ::= & \texttt{unit} \mid \texttt{string} \mid \texttt{bool} \\
 & \mid & [m_i{:}_{p_i} \tau_i \rightarrow_{p_i} \tau_i]^{1..n} \mid \tau\, \texttt{ref}_p \mid \texttt{stack} \\
 & & \\
v & ::= & \texttt{()} \mid \texttt{s} \mid \texttt{true} \mid \texttt{false} \\
 & & \\
e & ::= & v \mid x \mid e; e \mid \texttt{print}\; e \\
 & \mid & \texttt{if}\; e\; \texttt{then}\; e\; \texttt{else}\; e \\
 & \mid & \texttt{let}\; ds\; \texttt{in}\; e \\
 & \mid & e.m(e) \\
 & \mid & \texttt{!}\, e \mid e := e \\
 & \mid & \texttt{case}\; e\; \texttt{of}\; (pat \Rightarrow e \mid \_ \Rightarrow e) \\
 & & \\
pat & ::= & \texttt{nil} \mid \{o.m_i\}^{1..n}\, [x, y, n] :: pat \mid \_ :: pat \mid x \\
 & & \\
d & ::= & (\texttt{string}\; x = e) \\
 & \mid & (\texttt{bool}\; x = e) \\
 & \mid & (\texttt{ref}\; x = e) \\
 & \mid & (\texttt{object}\; o = [m_i : \tau_i \rightarrow \tau_i' = \varsigma\, x_i.\lambda y_i.e_i]^{1..n}) \\
 & & \\
ds & ::= & . \mid d\; ds \\
 & & \\
a & ::= & (\texttt{before}\; \{o.m_i\}^{1..n}(x, y, s, n) = e) \\
 & \mid & (\texttt{after}\; \{o.m_i\}^{1..n}(x, y, s, n) = e) \\
 & & \\
as & ::= & . \mid d\; as \mid a\; as \\
 & & \\
aspcts & ::= & . \mid p : \{as\}\; aspcts \\
 & & \\
prog & ::= & ds\; aspcts\; e
\end{array}
$$

Figure 11. Source Language Syntax

respectively. The string can be used when printing out debugging information, profiling information, etc.

Advice in the source language is either before advice that runs before a method call or after advice that runs after the method call. Similar to the source-language stack patterns, when the advice is triggered, x is bound to self, y is bound to the method argument, and n is bound to a string corresponding to the method 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 source into core.

3.2 Examples

Figure 12 and 13 display example code.[3] Figure 12 presents the (partial) definition of the `sys` object, which implements a number of system calls. We have shown some of the file operations here.

Figure 13 presents a couple of simple security policies we have implemented to exhibit the basic language features. Programmers would use these aspects to sandbox untrusted code [11, 18, 9, 3]. 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 `sys` object. 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

[3] These examples use a number of additional standard operations we have not formalized, but we have implemented in our system.

```
object sys = [
  openW : file   = x.y:string.
    if (x.exi (y))
    then (x.openO (y)) else (x.openC (y)),
  openA : file   = x.y:string. ...,
  openR : file   = x.y:string. ...,
  openO : file   = x.y:string. ...,
  openC : file   = x.y:string. ...,
  write : int    = x.y:file * string. ...,
  read  : string = x.y:file * int. ...,
  exists: bool   = x.y:string. ...,
  ...
  ]
```

Figure 12. System Object

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.*, `sys.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 `sys.openC` and `sys.openO` are "helper functions" that should only be called by `sys.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 `sys.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 A Case Study in Security

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 [10]. 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 [9]) 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 our language, testing them for harmlessness. We omitted those elements of the policies that were particularly Java-specific. 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. Some of the other policies we investigated include the following:

- **NoBashingExceptTmp** allows modification of files in the "tmp" directory but no others.

```
limitdirectories:{
  (string alloweddirectory = "tests/")
  (before {sys.openR, sys.openW, sys.openA, sys.openC,
           sys.openO} (x,y,s,n) =
    let
      (string directory =
         substring (y, 0, (lastindexof (y, "/")+1)))
    in
      (if (directory == alloweddirectory)
       then ()
       else ((print "Forbidden directory.\n");
             (abort ())) ))
}

limitcreate:{
  (ref filescreated:int = 0)
  (int createlimit = 10)
  (before {sys.openC} (x,y,s,n) =
    if ((!filescreated + 1) > createlimit)
    then ((print "Too many files created.\n");
          (abort ()))
    else ())
  (after {sys.openC} (x,y,s,n) =
    ((filescreated := (!filescreated) + 1); ()))
}

opencheck:{
  (before {sys.openC, sys.openO} (x,y,s,n) =
    case s of
    ( _::{sys.openW}[x,y,n]::tail => ()
    | _ => (print ("Invalid openW call.\n");
            abort ())))
}
```

Figure 13. Simple Security Aspects

- **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.
- **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.

3.4 Source Language Metatheory

To gives a semantics to our source language, we define a type-preserving transformation into the core language. More precisely, we define a type-preserving transformation into `Core2`, which 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 source language aspect is implemented as the appropriate core language state, objects and advice. Each new aspect is placed in its own new protection domain, which sits underneath the protection domain $main$ for the mainline code in the security lattice. 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 source language aspects is harmless.

Overall, the translation is defined as a collection of mutually recursive judgements that both type check the source and simultaneously translate it into the core. In general, the judgements have the form $P; \Gamma; p \vdash source : \tau \overset{\mathsf{ann}}{\Longrightarrow} e$ where P describes the program points that may trigger advice, Γ is a source typing context, p is the protection domain for the code, $source$ is source-language syntax to be translated, τ is its type, and e is the `Core2` languge code that results. The annotation `ann` above the arrow indicates the sort of source syntax that is being translated (*e.g.,* `dec` for declarations, `as` for aspect declarations, *etc.*,).

Due to space constraints, we cannot describe all of the details of the translation here (please see the appendix). However, we can show the one key rule that gives source-level aspects two interpretations in the core:

$$\frac{\begin{array}{c} P; \Gamma; p' \vdash as; .; () : \texttt{unit} \overset{\mathsf{as}}{\Longrightarrow} e' \\ P; \Gamma; main \vdash .; aspcts; e : \tau \overset{\mathsf{dec}}{\Longrightarrow} e'' \qquad \vdash p' < main \end{array}}{P; \Gamma; main \vdash .; p'{:}\{as\}\ aspcts; e : \tau \overset{\mathsf{dec}}{\Longrightarrow} p'\texttt{<}()|e'\texttt{>}; e''}$$

This rule translates one aspect ($p'{:}\{as\}$) in a sequence into a `Core2` expression. Assuming e' is the code that results from translating as, the declarations that make up the aspect, then the resulting `Core2` 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 `Core2` expressions.

THEOREM 3.1 (Translation Type Safety).
If $\vdash prog : \texttt{bool} \overset{\mathsf{prog}}{\Longrightarrow} e'$, *then* $.; main \vdash_2 e' : \texttt{bool}$.

The Translation Type Safety Theorem, when combined with properties of the core, gives us our final theorem establishing that the reference and implementation semantics of the source language coincide and therefore that aspects are harmless. The proof has a similar structure to the proof of non-interference for the core covered in Section 2.5.

THEOREM 3.2 (Source Language Aspects are Harmless).
If $\vdash prog : \texttt{bool} \overset{\mathsf{prog}}{\Longrightarrow} e'$
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 $v_1 = v_2$.

4. Related Work

Over the last several years, a number of researchers have begun to build semantic foundations for aspect-oriented programming paradigms [28, 8, 15, 16, 21, 26, 27]. This foundational work provides a starting point from which one can begin to analyze the properties of aspect-oriented programs, develop principled new programming features, study verification techniques and derive useful type systems. In this paper, our semantic foundations were derived directly from earlier work by Walker, Zdancewic and Ligatti [27]. The main novelty with respect to this earlier research is the development of a type system for ensuring that aspects do not interfere with each other or the mainline computation.

Clifton and Leavens [4] proposed techniques for Hoare-style reasoning about aspect-oriented programs using *assistants* and *observers*. Their notion of observers is similar to our conception of harmless advice — observers do not interfere with the mainline computation. However, the details of our type and effect system are entirely different from their Hoare logic. One point of interest is that Clifton and Leavens mention that it is not clear whether their model can "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.

Rinard, Salcianu, and Bugrara [25] developed a system that classifies 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's orthogonal, independent, and observation interactions. Rinard's tool uses a variety of data flow and control flow analyses and operates within Java's nominal type structure, so the details of their system are quite different from our own. In addition, Rinard's system is described informally in English; he has not proven any properties of his analyses.

Douence, Fradet and Südholt [7] analyze aspects defined by recursion together with parallel and sequencing combinators. They develop a number of formal laws for reasoning about their combinators and an algorithm that is able to detect *strong independence*. Two pieces of advice are strongly independent when they do 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 is determined exclusively by analysis of the point cut designators of the two pieces of advice and consequently it is orthogonal to our analysis which (mostly) ignores the point cuts and examines the advice bodies instead. It would be interesting to explore how to put these two different ideas together.

Krishnamurthi, Fisler and Greenberg [19] tackle 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 point cut designators, they verify programs using model checking. Their approach to verification is partly modular since as long as the set of point cuts does not change and the underlying mainline code remains fixed, it is not necessary to reanalyze the mainline code as advice definitions are edited. However, if the pointcuts or mainline code do 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 point cut 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). Having said that, it certainly seems possible to combine these two different ideas in a single system, an interesting idea for future research.

Another interesting line of current research involves finding ways to add aspect-oriented programming features to languages with module systems, or vice-versa. One of the first systems to combine aspects and modules effectively was Lieberherr, Lorenz and Ovlinger's *Aspectual Collaborations* [20, 23]. Their proposal allows module programmers to choose the join points (i.e., control-flow points) that they will expose to external advice. External advice cannot intercept control-flow points that have not been exposed. Aldrich [2] has proposed another model for combining aspects and modules called *Open Modules*. Open Modules have a special module sealing operator that hides internal control-flow points from external advice. Aldrich has used logical relations to show that sealed modules have a powerful implementation-independence property [1]. The current report differs from this previous research as it does not suggest that visibility of the interception points be limited; instead, we suggest limiting the capabilities of advice. Once again, it seems quite likely that one could combine both ideas.

Finally, together with Washburn and Weirich [6], we have shown how to extend the core calculus and a related functional source language with polymorphic functions, polymorphic advice and run-time type analysis. In the future, we are planning to merge these two efforts in order to support safe but flexible type-directed aspect-oriented programming.

5. Conclusions

In this paper, we have investigated the idea of *harmless advice*: aspect-oriented advice that does not interfere with the mainline computation. While strictly less powerful than ordinary advice, we believe that harmless advice can be used in many contexts including security monitoring, profiling, logging, and for some debugging tasks. Harmless advice has the advantage that it may be added to a program after-the-fact, in the typical aspect-oriented style, yet programmers do not have to worry about it corrupting important mainline data invariants and hence they retain the ability to perform local reasoning about partial correctness of their programs.

References

[1] J. Aldrich. Open modules: A proposal for modular reasoning in aspect-oriented programming. In *Workshop on foundations of aspect-oriented languages*, Mar. 2004.

[2] J. Aldrich. Open modules: Reconciling extensibility and information hiding. In *Proceedings of the Software Engineering Properties of Languages for Aspect Technologies*, Mar. 2004.

[3] L. Bauer, J. Ligatti, and D. Walker. Composing security policies in polymer. In *ACM Conference on Programming Language Design and Implementation*, June 2005.

[4] C. Clifton and G. T. Leavens. Assistants and observers: A proposal for modular aspect-oriented reasoning. In *Foundations of Aspect Languages*, Apr. 2002.

[5] 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.

[6] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. PolyAML: A polymorphic aspect-oriented functional programmming language. In *ACM International Conference on Functional Programming*, Sept. 2005. To appear.

[7] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150, New York, NY, USA, 2004. ACM Press.

[8] R. Douence, O. Motelet, and M. Südholt. A formal defi nition of crosscuts. In *Third International Conference on Metalevel architectures and separation of crosscutting concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Berlin, Sept. 2001. Springer-Verlag.

[9] 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.

[10] D. Evans. *Policy-Directed Code Safety*. PhD thesis, MIT, 1999.

[11] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Security and Privacy*, Oakland, CA, May 1999.

[12] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantifi cation and obliviousness. In *Workshop on Advanced Separation of Concerns*, Oct. 2000.

[13] 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.

[14] List of main users. AspectJ Users List: aspectj-users@eclipse.org, June 2004. Requires subscription to access archives.

[15] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of typed aspect-oriented programs. Unpublished manuscript., 2003.

[16] 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.

[17] 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.

[18] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specifi ed monitoring of temporal properties. In *European Conference on Real-time Systems*, York, UK, June 1999.

[19] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *Foundations of Software Engineering*, Oct.-Nov. 2004.

[20] K. J. Lieberherr, D. Lorenz, and J. Ovlinger. Aspectual collaborations – combining modules and aspects. *The Computer Journal*, 46(5):542–565, September 2003.

[21] 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.

[22] A. Myers and B. Liskov. Jflow: Practical mostly-static information flow control. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 226–241, Jan. 1998.

[23] J. Ovlinger. *Modular Programming with Aspectual Collaborations*. PhD thesis, Northeastern University, 2003.

[24] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, Jan. 2003.

[25] M. Rinard, A. Salcianu, and S. Bugrara. A classifi cation system and analysis for aspect-oriented programs. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 147–158, New York, NY, USA, 2004. ACM Press.

[26] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 158–167, 2003.

[27] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ACM International Conference on Functional Programming*, Uppsala, Sweden, Aug. 2003.

[28] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In G. T. Leavens and R. Cytron, editors, *Foundations of Aspect-Oriented Languages Workshop*, pages 17–25, Apr. 2002. Iowa State University University technical report 02-06.

Acknowledgments

We would like to thanks Geoff Washburn for suggesting we translate the source language in `Core2` as a way to establish source language aspects are harmless. We would also like to thank Geoff, Stephanie Weirich and Steve Zdancewic for stimulating discussions on aspects and related topics.

A. Source Language

A.1 Syntax

$$
\begin{array}{rcl}
\tau & ::= & \texttt{unit} \mid \texttt{string} \mid \texttt{bool} \\
 & \mid & [m_i{:}_{p_i} \tau_i \rightarrow_{p_i} \tau_i]^{1..n} \mid \tau\ \texttt{ref}_p \mid \texttt{stack} \\
 & & \\
v & ::= & \texttt{()} \mid \texttt{s} \mid \texttt{true} \mid \texttt{false} \\
 & & \\
e & ::= & v \mid x \mid e;e \mid \texttt{print}\ e \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & \mid & \texttt{let}\ ds\ \texttt{in}\ e \\
 & \mid & e.m(e) \\
 & \mid & \texttt{!}\ e \mid e := e \\
 & \mid & \texttt{case}\ e\ \texttt{of}\ \texttt{(}pat \Rightarrow e \mid \_ \Rightarrow e\texttt{)} \\
 & & \\
pat & ::= & \texttt{nil} \mid \{o.m_i\}^{1..n}\,\texttt{[}x,y,n\texttt{]} :: pat \mid \_ :: pat \mid x \\
 & & \\
d & ::= & \texttt{(string}\ x = e\texttt{)} \\
 & \mid & \texttt{(bool}\ x = e\texttt{)} \\
 & \mid & \texttt{(ref}\ x = e\texttt{)} \\
 & \mid & \texttt{(object}\ o = [m_i : \tau_i \rightarrow \tau_i' = \varsigma\, x_i.\lambda y_i.e_i]^{1..n}\texttt{)} \\
 & & \\
ds & ::= & . \mid d\ ds \\
 & & \\
a & ::= & \texttt{(before}\ \{o.m_i\}^{1..n}(x,y,s,n) = e\texttt{)} \\
 & \mid & \texttt{(after}\ \{o.m_i\}^{1..n}(x,y,s,n) = e\texttt{)} \\
 & & \\
as & ::= & . \mid d\ as \mid a\ as \\
 & & \\
aspcts & ::= & . \mid p : \{as\}\ aspcts \\
 & & \\
prog & ::= & ds\ aspcts\ e
\end{array}
$$

A.2 Translation from Source into Core

Please see the following pages.

$$\boxed{P;\Gamma \vdash v : \tau \overset{\textsf{val}}{\Longrightarrow} v'}$$

$$\frac{}{P;\Gamma \vdash () : \texttt{unit} \overset{\textsf{val}}{\Longrightarrow} ()} \qquad \frac{}{P;\Gamma \vdash s : \texttt{string} \overset{\textsf{val}}{\Longrightarrow} s}$$

$$\frac{}{P;\Gamma \vdash \texttt{true} : \texttt{bool} \overset{\textsf{val}}{\Longrightarrow} \texttt{true}} \qquad \frac{}{P;\Gamma \vdash \texttt{false} : \texttt{bool} \overset{\textsf{val}}{\Longrightarrow} \texttt{false}}$$

$$\boxed{P;\Gamma;p \vdash e : \tau \overset{\textsf{exp}}{\Longrightarrow} e'}$$

$$\frac{P;\Gamma \vdash v : \tau \overset{\textsf{val}}{\Longrightarrow} v'}{P;\Gamma;p \vdash v : \tau \overset{\textsf{exp}}{\Longrightarrow} v'} \qquad \frac{\Gamma(x) = \tau}{P;\Gamma;p \vdash x : \tau \overset{\textsf{exp}}{\Longrightarrow} x} \qquad \frac{P;\Gamma;p \vdash e_1 : \texttt{unit} \overset{\textsf{exp}}{\Longrightarrow} e_1' \quad P;\Gamma;p \vdash e_2 : \tau \overset{\textsf{exp}}{\Longrightarrow} e_2'}{P;\Gamma;p \vdash e_1; e_2 : \tau \overset{\textsf{exp}}{\Longrightarrow} e_1'; e_2'}$$

$$\frac{P;\Gamma;p \vdash e : \texttt{string} \overset{\textsf{exp}}{\Longrightarrow} e'}{P;\Gamma;p \vdash \texttt{print}\ e : \texttt{unit} \overset{\textsf{exp}}{\Longrightarrow} \texttt{print}\ e'} \qquad \frac{P;\Gamma;p \vdash e_1 : \texttt{bool} \overset{\textsf{exp}}{\Longrightarrow} e_1' \quad P;\Gamma;p \vdash e_2 : \tau \overset{\textsf{exp}}{\Longrightarrow} e_2' \quad P;\Gamma;p \vdash e_3 : \tau \overset{\textsf{exp}}{\Longrightarrow} e_3'}{P;\Gamma;p \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : \tau \overset{\textsf{exp}}{\Longrightarrow} \texttt{if}\, e_1'\, \texttt{then}\, e_2'\, \texttt{else}\, e_3'}$$

$$\frac{P;\Gamma;p \vdash ds; .; e : \tau \overset{\textsf{dec}}{\Longrightarrow} e'}{P;\Gamma;p \vdash \texttt{let}\ ds\ \texttt{in}\ e : \tau \overset{\textsf{exp}}{\Longrightarrow} e'} \qquad \frac{\begin{array}{ll} P;\Gamma;p \vdash e_1 : [m_i{:}_{p_i}\tau_i]^{1..n} \overset{\textsf{exp}}{\Longrightarrow} e_1' & \tau_j = \tau \to_p \tau' \\ P;\Gamma;p \vdash e_2 : \tau \overset{\textsf{exp}}{\Longrightarrow} e_2' & \vdash p_j = p \end{array}}{P;\Gamma;p \vdash e_1.m_j(e_2) : \tau' \overset{\textsf{exp}}{\Longrightarrow} e_1'.m_j\ e_2'}$$

$$\frac{P;\Gamma;p \vdash e : \tau\ \texttt{ref}_{p'} \overset{\textsf{exp}}{\Longrightarrow} e' \quad \vdash p \le p'}{P;\Gamma;p \vdash !\ e : \tau \overset{\textsf{exp}}{\Longrightarrow} !\ e'} \qquad \frac{P;\Gamma;p \vdash e_1 : \tau\ \texttt{ref}_{p'} \overset{\textsf{exp}}{\Longrightarrow} e_1' \quad P;\Gamma;p \vdash e_2 : \tau \overset{\textsf{exp}}{\Longrightarrow} e_2' \quad \vdash p' \le p}{P;\Gamma;p \vdash e_1 := e_2 : \texttt{unit} \overset{\textsf{exp}}{\Longrightarrow} e_1' := e_2'}$$

$$\frac{P;\Gamma;p \vdash e_1 : \texttt{stack} \overset{\textsf{exp}}{\Longrightarrow} e_1' \quad P;p \vdash pat \overset{\textsf{pat}}{\Longrightarrow} pat' \dashv \Gamma';\Theta \quad P;\Gamma,\Gamma';p \vdash e_2 : \tau \overset{\textsf{exp}}{\Longrightarrow} e_2' \quad P;\Gamma;p \vdash e_3 : \tau \overset{\textsf{exp}}{\Longrightarrow} e_3'}{P;\Gamma;p \vdash \texttt{case}\ e_1\ \texttt{of}\ (pat \Rightarrow e_2 \mid \_ \Rightarrow e_3) : \tau \overset{\textsf{exp}}{\Longrightarrow} \texttt{case}\ e_1'\ \texttt{of}\ (pat' \Rightarrow \texttt{split}(\Theta, e_2') \mid \_ \Rightarrow e_3')}$$

$$\boxed{\texttt{split}(\Theta, e)}$$

$$\texttt{split}(\cdot, e) = e \qquad \texttt{split}(a \to (x,y,z), \Theta) = \texttt{split}(\Theta, \texttt{split}\,(x,y,z) = a\ \texttt{in}\ e)$$

$$\boxed{\Gamma \dashv \Theta \Rightarrow \Gamma'}$$

$$\frac{}{\Gamma \dashv \cdot \Rightarrow \Gamma} \qquad \frac{\Gamma \dashv \Theta \Rightarrow \Gamma'}{\Gamma, x:\tau, y:\tau', z:\tau'' \dashv \Theta, z \to (x,y,z) \Rightarrow \Gamma', z : (\tau \times \tau' \times \tau'')}$$

$$\boxed{P;p \vdash pat \overset{\textsf{pat}}{\Longrightarrow} pat' \dashv \Gamma';\Theta}$$

$$\frac{(P(o.m_i) = (\tau_{self}, \tau_{arg}, \tau_{res}, p_i))^{(1\le i\le n)} \quad P;p \vdash pat \overset{\textsf{pat}}{\Longrightarrow} pat' \dashv \Gamma';\Theta}{\begin{array}{c} P;p \vdash \{o.\vec{m}\}[x,y,n] :: pat \overset{\textsf{pat}}{\Longrightarrow} \{o\vec{m}_{\textsf{pre}}\}_p[z] :: pat' : \\ (\Gamma', x:\tau_{self}, y:\tau_{arg}, n:\texttt{string}; \Theta, z \to (x,y,n)) \end{array}}$$

$$\frac{}{P;p \vdash nil \overset{\textsf{pat}}{\Longrightarrow} nil \dashv \cdot;\cdot} \qquad \frac{P;p \vdash pat \overset{\textsf{pat}}{\Longrightarrow} pat' \dashv \Gamma';\Theta}{P;p \vdash \_ :: pat \overset{\textsf{pat}}{\Longrightarrow} \_ :: pat' \dashv \Gamma';\Theta} \qquad \frac{}{P;p \vdash x \overset{\textsf{pat}}{\Longrightarrow} x \dashv \cdot, (x:\texttt{stack});\cdot}$$

Figure 14. Translation: Part 1

$$\boxed{P;\Gamma;p \vdash as; aspcts; e : \tau \overset{\textsf{dec}}{\Longrightarrow} e'}$$

$$\frac{\begin{array}{c} P;\Gamma;p' \vdash as; .; () : \texttt{unit} \overset{\textsf{as}}{\Longrightarrow} e' \\ P;\Gamma;main \vdash .; aspcts; e : \tau \overset{\textsf{dec}}{\Longrightarrow} e'' \qquad \vdash p' < main \end{array}}{P;\Gamma;main \vdash .; p'\!:\!\{as\}\ aspcts; e : \tau \overset{\textsf{dec}}{\Longrightarrow} p'\texttt{<}()|e'\texttt{>}; e''} \qquad \frac{P;\Gamma;p \vdash e : \tau \overset{\textsf{exp}}{\Longrightarrow} e'}{P;\Gamma;p \vdash .; .; e : \tau \overset{\textsf{dec}}{\Longrightarrow} e'}$$

$$\frac{\begin{array}{c} P;\Gamma;p \vdash e_1 : \texttt{string} \overset{\textsf{exp}}{\Longrightarrow} e_1' \\ P;\Gamma, x : \texttt{string}; p \vdash as; aspcts; e_2 : \tau \overset{\textsf{dec}}{\Longrightarrow} e_2' \end{array}}{\begin{array}{c} P;\Gamma;p \vdash (\texttt{string}\, x = e_1)\, as; aspcts; e_2 : \tau \overset{\textsf{dec}}{\Longrightarrow} \\ \texttt{let}\ x = e_1'\ \texttt{in}\ e_2' \end{array}} \qquad \frac{\begin{array}{c} P;\Gamma;p \vdash e_1 : \texttt{bool} \overset{\textsf{exp}}{\Longrightarrow} e_1' \\ P;\Gamma, x : \texttt{bool}; p \vdash as; aspcts; e_2 : \tau \overset{\textsf{dec}}{\Longrightarrow} e_2' \end{array}}{\begin{array}{c} P;\Gamma;p \vdash (\texttt{bool}\, x = e_1)\, as; aspcts; e_2 : \tau \overset{\textsf{dec}}{\Longrightarrow} \\ \texttt{let}\ x = e_1'\ \texttt{in}\ e_2' \end{array}}$$

$$\frac{\begin{array}{c} (P;(\Gamma, x : \tau_{self}, y : \tau_i); p \vdash e_i : \tau_i' \overset{\textsf{exp}}{\Longrightarrow} e_i')^{1 \le i \le n} \\ (P,(o.m_i : (\tau_{self}, \tau_i, \tau_i', p))^{1..n}); (\Gamma, o : \tau_{self}); p \vdash as; aspcts; e_2 : \tau \overset{\textsf{dec}}{\Longrightarrow} e_2' \end{array}}{\begin{array}{rl} \multicolumn{2}{l}{P;\Gamma;p \vdash (\texttt{object}\, o = [m_i : \tau_i \to \tau_i' = \varsigma\, x_i.\lambda y_i.e_i]^{1..n})\, as; aspcts; e_2 : \tau \overset{\textsf{dec}}{\Longrightarrow}} \\ \texttt{let}\ om_{1,\texttt{pre}} = \texttt{new}_p : (\tau_{self}, \tau_1, \texttt{string})\ \texttt{in} & \ldots\ \texttt{let}\ om_{n,\texttt{pre}} = \texttt{new}_p : (\tau_{self}, \tau_n, \texttt{string})\ \texttt{in} \\ \texttt{let}\ om_{1,\texttt{post}} = \texttt{new}_p : (\tau_{self}, \tau_1', \texttt{string})\ \texttt{in} & \ldots\ \texttt{let}\ om_{n,\texttt{post}} = \texttt{new}_p : (\tau_{self}, \tau_n', \texttt{string})\ \texttt{in} \\ \texttt{let}\ o = [m_i = \varsigma_p\, x_i.\lambda_p y_i : \tau_i. & \texttt{store}\ om_{i,\texttt{pre}}\, [(x_i, y_i, \text{“}o.m_i''\text{”})]\ \texttt{in} \\ & om_{i,\texttt{pre}}\, [(x_i, y_i, \text{“}o.m_i\text{“})]; \\ & \texttt{let}\ res_i = e_i'\ \texttt{in} \\ & om_{i,\texttt{post}}\, [(x_i, res_i, \text{“}o.m_i\text{“})]; \\ & res_i \\ ]^{1..n}\ \texttt{in}\ e_2' & \end{array}}$$

$$\texttt{where}\ \tau_{self} \ = \ [m_i{:}_p\tau_i \to_p \tau_i']^{1..n}$$

$$\frac{P;\Gamma;p \vdash e_1 : \tau \overset{\textsf{exp}}{\Longrightarrow} e_1' \quad P;\Gamma, x : \tau\ \texttt{ref}_p; p \vdash as; aspcts; e_2 : \tau' \overset{\textsf{dec}}{\Longrightarrow} e_2'}{P;\Gamma;p \vdash (\texttt{ref}\, x = e_1)\, as; aspcts; e_2 : \tau' \overset{\textsf{dec}}{\Longrightarrow} \texttt{let}\ x = \texttt{ref}_p\ e_1'\ \texttt{in}\ e_2'}$$

$$\frac{\begin{array}{c} (P(o.m_i) = (\tau_{self}, \tau_{arg}, \tau_{res}, p_i))^{(1 \le i \le n)} \\ P;(\Gamma, x : \tau_{self}, y : \tau_{arg}, s : \texttt{stack}, n : \texttt{string}); p \vdash e_1 : \texttt{unit} \overset{\textsf{exp}}{\Longrightarrow} e_1' \quad P;\Gamma;p \vdash as; aspcts; e_2 : \tau \overset{\textsf{dec}}{\Longrightarrow} e_2' \end{array}}{\begin{array}{rl} \multicolumn{2}{c}{P;\Gamma;p \vdash (\texttt{before}\ \{o.\vec{m}\}(x, y, s, n) = e_1)\, as; aspcts; e_2 : \tau \overset{\textsf{dec}}{\Longrightarrow}} \\ \Uparrow \{\{o\vec{m}_{\texttt{pre}}\}_p.z \to_p & \texttt{split}\,(x, y, n) = z\ \texttt{in} \\ & \texttt{let}\ s = \texttt{stack}()\ \texttt{in} \\ & e_1'\}; e_2' \end{array}}$$

$$\frac{\begin{array}{c} (P(o.m_i) = (\tau_{self}, \tau_{arg}, \tau_{res}, p_i))^{(1 \le i \le n)} \\ P;(\Gamma, x : \tau_{self}, y : \tau_{res}, s : \texttt{stack}, n : \texttt{string}); p \vdash e_1 : \texttt{unit} \overset{\textsf{exp}}{\Longrightarrow} e_1' \quad P;\Gamma;p \vdash as; aspcts; e_2 : \tau \overset{\textsf{dec}}{\Longrightarrow} e_2' \end{array}}{\begin{array}{rl} \multicolumn{2}{c}{P;\Gamma;p \vdash (\texttt{after}\ \{o.\vec{m}\}(x, y, s, n) = e_1)\, as; aspcts; e_2 : \tau \overset{\textsf{dec}}{\Longrightarrow}} \\ \Uparrow \{\{o\vec{m}_{\texttt{post}}\}_p.z \to_p & \texttt{split}\,(x, y, n) = z\ \texttt{in} \\ & \texttt{let}\ s = \texttt{stack}()\ \texttt{in} \\ & e_1'\}; e_2' \end{array}}$$

$$\boxed{\vdash prog : \tau \overset{\textsf{prog}}{\Longrightarrow} e'}$$

$$\frac{.; .; main \vdash ds; aspcts; e : \tau \overset{\textsf{dec}}{\Longrightarrow} e'}{\vdash ds\ aspcts\ e : \tau \overset{\textsf{prog}}{\Longrightarrow} e'}$$

Figure 15. Translation: Part II