A Theory of Aspects

David Walker*
Princeton University
dpw@cs.princeton.edu

Steve Zdancewic
University of Pennsylvania
stevez@cis.upenn.edu

Jay Ligatti
Princeton University
jligatti@cs.princeton.edu

Abstract

This paper define the semantics of MinAML, an idealized aspect-oriented programming language, by giving a type-directed translation from its user-friendly external language to its compact, well-defined core language. We argue that our framework is an effective way to give semantics to aspect-oriented programming languages in general because the translation eliminates shallow syntactic differences between related constructs and permits definition of a clean, easy-to-understand, and easy-to-reason-about core language.

The core language extends the simply-typed lambda calculus with two central new abstractions: explicitly labeled program points and first-class advice. The labels serve both to trigger advice and to mark continuations that the advice may return to. These constructs are defined orthogonally to the other features of the language and we show that our abstractions can be used in both functional and object-oriented contexts. The labels are well-scoped and the language as a whole is well-typed. Consequently, programmers can use lexical scoping in the standard way to prevent aspects from interfering with local program invariants.

Categories and Subject Descriptors

D.3.1 [**Formal Definitions and Theory**]: Semantics; D.3.3 [**Language Constructs and Features**]: Control structures; F.3.2 [**Semantics of Programming Languages**]: Operational semantics

General Terms

Languages, Design, Theory

Keywords

Aspects, Aspect-oriented Programming, Operational Semantics, Type Theory

*This research was supported in part by National Science Foundation CAREER grant No. CCR-0238328.

ICFP'03, August 25–29, 2003, Uppsala, Sweden.

1. Introduction

Aspect-oriented programming languages (AOPL) [3], such as AspectJ [10] and Hyper/J [13], provide the facility to intercept the flow of control in an application and insert new computation at that point. In this approach, certain control-flow points, called *join points*, are designated as special—typically, join points include the entry and exit points of functions. Computation at these control flow points may be intercepted by a piece of *advice*, which is a piece of code that can manipulate the surrounding local state or cause global effects. Advice is triggered only when the run-time context at a join point meets programmer-specified conditions, making advice a useful way to instrument programs with debugging information, performance monitors, or security checks. An *aspect* is a collection of advice and corresponding join points that apply to a particular program.

Much of the research on aspect-oriented programming has focused on applying aspects in various problem domains and on integration of aspects into full-scale programming languages such as Java. However, aspects are a very powerful and complex language mechanism, combining features of both dynamic scoping and continuation manipulation. While recent research efforts [6, 9, 14, 16] have made significant progress on understanding some of the semantic issues involved, the theoretical underpinnings for this novel paradigm lag well behind practical implementation efforts.

The primary goal of this paper is to distill aspect-oriented programming into its fundamental components: (1) a means of designating "interesting" control-flow points, and (2) a way of manipulating the data and computation at those points. The goal is to obtain a clear, reusable semantic framework, but there are at least two significant difficulties:

1. Most object-oriented AOPL specify that there are join points at the entry- and the exit-points of every method. Such a definition of join points breaks the principle of *orthogonality*, which suggests that each programming language construct should be understood independently of other programming language constructs. Tightly coupling join-point definition with the semantics of methods and objects makes it impossible to understand aspects without first understanding methods and objects, which are complicated in isolation.

2. There are several different and relatively complex varieties of advice one can give to any particular join point. For instance, AspectJ allows programmers to specify advice *before*, *after* and *around* its various join points. While each sort of advice has a similar "feel,"

they are sufficiently different that they appear to require independent semantic analysis.

To resolve these difficulties, we adopt the central ideas of a type-theoretic semantic framework defined by Harper and Stone for Standard ML [8]. Rather than give a semantics directly to a large and relatively complex AOPL with several different kinds of advice and join points, we translate the unwieldy *external language* into a simpler *core language* and then provide a precise and elegant operational semantics for the core. The translation eliminates shallow syntactic differences between similar constructs, thereby shrinking the number of features in the core and effectively modularizing the overall semantics. We also use the translation as an implementation strategy for the source language.

This core language defines two central new abstractions.

1. Explicit, labeled join points that are defined orthogonally from the other constructs in the language, and
2. A single kind of first-class advice that, together with labeled join points, gives meaning to the before, after and around advice that one finds in AspectJ.

The main contributions of this paper include:

- A type-theoretic interpretation of an idealized, but useful aspect-oriented language called MinAML that includes advice, functions, and objects.
- A minimalist core aspect language with a well-defined operational interpretation, and a sound type system.
- Evidence that our core language is general, expressive, and scalable. The paper gives a number of examples written in our aspect framework. It also shows that enriching the point-cut language with context-sensitive predicates and adding features such as objects does not change the central machinery needed for aspects.
- A prototype implementation of a toy functional, aspect-oriented language, AspectML (AML) that includes advice and point-cut declarations. The implementation comes in two parts, mirroring our theoretical development. The core language is implemented as a library for SML/NJ, and AML is translated into SML by a simple rewriter that inserts calls to the core library.

The next section introduces the features of the core aspect calculus and its syntax, largely via examples. These examples motivate the design of the operational semantics and type system, which are described in Sections 2.1 and 2.2. Section 3 defines the external language, MinAML. Subsequent sections generalize the core calculus and MinAML by extending them to include objects (Section 3.2) and richer point-cut designators (Section 4). The paper concludes with a description of a prototype implementation in SML/NJ and discussion of related and future work (Sections 5 and 6).

2. Core aspect calculus

Labeled join points $l\langle e\rangle$ are the essential mechanism of the core aspect calculus. The labels, which are drawn from some infinite set of identifiers, serve several purposes: They mark the points at which advice may be triggered, they provide the appropriate contextual information for trigger predicates, and they mark points to which control may be transferred when some advice decides to abort part of the current computation. For example, in the expression v_1 + $l\langle e_2\rangle$, after e_2 has been evaluated to a value v_2, evaluation of the resulting subterm $l\langle v_2\rangle$ causes any advice associated with the label l to be triggered. This construct permits the unambiguous marking of *any* control flow point rather than relying upon some *a priori* designation of the "interesting control flow points," which are hard-wired in most aspect-oriented languages.

Advice, at the most fundamental level, is a computation that exchanges data with a particular join point, and hence a piece of advice is similar to a function. However, there are some subtleties involved in the definition. Advice can not only manipulate the data at the point, it can also influence the control flow—perhaps by skipping code that would have been run in the advice's absence.

The advice $\{l.x \to e\}$ indicates that it will be triggered when control flow reaches a point labeled l. The variable x is bound to the data at that point, and evaluation proceeds with the expression e, the body of the advice. Assuming that the advice $\{l.x \to e\}$ has been installed in the program's dynamic environment, the example v_1 + $l\langle v_2\rangle$ evaluates to v_1 + $e\{v_2/x\}$. Note that the advice computes a value of the same type as its argument, in this case an integer—importantly, advice can be composed with other advice.

The same label may be used to tag distinct control flow points, as long as those points indicate computations of the same type. For example, the program $l\langle v_1\rangle$ + $l\langle v_2\rangle$ causes two instances of the advice $\{l.x \to e\}$ to be run, but one instance will be passed v_1 and the other will be passed v_2

Multiple pieces of advice may apply at the same control-flow point. Because, in general, advice may have effects, the order in which they run is important. It therefore seems natural that there should be at least two ways to install advice in the run-time environment, one that runs the new advice prior to any other and one that runs it after any other. (One could imagine generalizations of this idea, but this simple scheme suffices for many interesting applications of aspects.) Accordingly, the core aspect language includes expression forms a << e and a >> e to respectively install the advice a prior to and after the other advice. In both cases running the advice a is delayed until the corresponding join point is reached; the program continues as expression e.

The following examples show how advice precedence works (assuming that there is no other advice associated with label l in the environment).[1]

$$\{l.x \to x+1\} \texttt{ << } \{l.y \to y*2\} \texttt{ << } l\langle 3\rangle \quad \longmapsto^* \quad 7$$
$$\{l.x \to x+1\} \texttt{ << } \{l.y \to y*2\} \texttt{ >> } l\langle 3\rangle \quad \longmapsto^* \quad 8$$

Because it can be difficult to reason about the behavior of a program when the advice associated with a label is unknown, it is useful to introduce a scoping mechanism for labels. The expression `new` p:t. e allocates a fresh label that is bound to the variable p in the expression e. Labels are considered first class values, so the example above can be rewritten as follows:

$$\texttt{new } p\texttt{:int. } \{p.x \to x+1\} \texttt{ << } \{p.y \to y*2\} \texttt{ << } p\langle v\rangle$$

This ensures that only the advice explicitly declared in the scope of the `new` get triggered at the location $p\langle v\rangle$. The

[1] The operators << and >> are left-associative, and evaluation proceeds from left to right. Hence, a_1 << a_2 << e installs a_1 in the environment first and a_2 in the environment second, before proceeding with evaluation of e.

variables bound by the `new` expression α-vary, providing for modular program design.

With the features described so far, it is easy to see that aspects are a powerful (and potentially dangerous) tool. Consider the following example:

$$\texttt{new}\ p\texttt{:bool.}\ \{p.x \to p\langle x\rangle\} \ll p\langle\texttt{true}\rangle$$

This program immediately goes into an infinite loop, even though the underlying program to which the advice applies, `true`, is already a value. Wand and others [16] have observed that aspects can be used to implement arbitrary fixpoints of functions using this technique. As another example of the power of aspects, the program below shows how to encode a (somewhat inefficient) implementation of reference cells using the state provided by advice. A reference cell is represented as a pair of functions, the first dereferences the cell and the second updates the cell's contents. The data is stored in advice associated with label *ref*; the last advice to be run returns the current contents of the reference.

$$\begin{array}{l}\texttt{makeref} \stackrel{\text{def}}{=}\\ \quad \lambda \mathit{init}\texttt{:t. new}\ \mathit{ref}\texttt{:t.}\\ \quad \{\mathit{ref}.x \to \mathit{init}\} \ll\\ \quad \texttt{let}\ \mathit{get} = \lambda\_\texttt{:unit.}\mathit{ref}\langle\mathit{init}\rangle\ \texttt{in}\\ \quad \texttt{let}\ \mathit{set} = \lambda y'\texttt{:t.}\{\mathit{ref}.y \to y'\} \gg ()\ \texttt{in}\ (\mathit{get},\mathit{set})\end{array}$$

As these examples show, aspects can radically alter the semantics of a given programming language. Part of the contribution of this work is to provide a framework that makes studying these issues straightforward.

It is sometimes desirable for advice to suppress the execution of a piece of code or replace it altogether. The last feature of the core aspect calculus, written `return` v `to` l, allows such alterations to the control flow of the program. Operationally, `return` is very similar to throwing a value to a continuation or raising an exception. The value v is directly passed to the nearest enclosing control-flow point labeled l, bypassing any intervening pending computation. If there is no point with label l, the program halts with an error (this is analogous to an uncaught exception). As an example, the following program evaluates to the value 3:

$$\texttt{new}\ p\texttt{:int.}\ p\langle 4 + (\texttt{return}\ 3\ \texttt{to}\ p)\rangle$$

A second example (below) shows how to instrument a function $f = \lambda x : \texttt{bool}.\ e$ of type $\texttt{bool} \to t$ so that if its argument is true then e proceeds as usual, otherwise some alternative code e' is run.

$$\begin{array}{l}\texttt{new}\ f_{\texttt{pre}}\texttt{: bool.}\\ \texttt{new}\ f_{\texttt{post}}\texttt{: bool.}\\ \{f_{\texttt{pre}}.x \to \texttt{if}\ x\ \texttt{then}\ x\ \texttt{else return}\ e'\ \texttt{to}\ f_{\texttt{post}}\}\\ \gg\\ \lambda x\texttt{:bool.}f_{\texttt{post}}\langle\texttt{let}\ x = f_{\texttt{pre}}\langle x\rangle\ \texttt{in}\ e\rangle\end{array}$$

The strategy is to use two labels, $f_{\texttt{pre}}$ and $f_{\texttt{post}}$, that get triggered at the function's entry and exit. The advice associated with the precondition checks the value of x and, if it is `true` simply returns control to the body of the function. If x is false, the advice runs e' and returns the result directly to the point labeled $f_{\texttt{post}}$. The function is instrumented by adding the label $f_{\texttt{pre}}$, which will trigger the precondition advice to inspect the function argument x, and by adding the label $f_{\texttt{post}}$ around the entire function body, which specifies the return point from the function.

2.1 Operational Semantics

This section describes the operational semantics for the core language, whose grammar is summarized below. For simplicity, the base language is chosen to be the simply-typed lambda calculus with Booleans and n-tuples.

$$\begin{array}{lcl} l & \in & \text{Labels}\\ v & ::= & \{v.x \to e\} \mid b \mid l \mid \lambda x\texttt{:}t.\ e \mid (\vec{v})\\ e & ::= & x \mid v \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid e_1\ e_2\\ & \mid & (e_1,\ldots,e_n)^{(n\geq 0)} \mid \texttt{let}(\vec{x}\texttt{:}\vec{t}) = e_1\ \texttt{in}\ e_2\\ & \mid & \texttt{new}\ x\texttt{:}t.\ e \mid e_1\langle e_2\rangle \mid \texttt{return}\ e_1\ \texttt{to}\ e_2\\ & \mid & \{e_1.x \to e_2\} \mid e_1 \gg e_2 \mid e_1 \ll e_2\end{array}$$

Let b range over the Boolean values `true` and `false` and a range over advice values $\{v.x \to e_2\}$. The other syntactic categories in the language include labels for control-flow points (l), values (v) and expressions (e). If $\vec{e}$ is a vector of expressions e_1, e_2, ..., e_n for $n \geq 0$, then $(\vec{e})$ creates a tuple. The expression $\texttt{let}(\vec{x}:\vec{t}) = e_1\ \texttt{in}\ e_2$ binds the components of a tuple e_1 to the vector of variables $\vec{x}$ in the scope of e_2. Types on `let`-bound variables are often omitted when they are clear from context.

The point cut language has been reduced to the barest minimum for the core calculus. However, the language design and semantics are completely compatible with more expressive point cuts; Section 4 investigates several alternatives. Note that point cuts, advice and labels are first-class values; these values may be passed to and from functions just as any other data structure.

The operational semantics uses evaluation contexts (E) defined according to the following grammar:

$$\begin{array}{lcl} E & ::= & [\,] \mid \texttt{if}\ E\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid E\ e \mid v\ E\\ & \mid & (\vec{v}, E, \vec{e}) \mid E \ll e \mid E \gg e \mid E\langle e\rangle \mid l\langle E\rangle\\ & \mid & \{E.x \to e\} \mid \texttt{return}\ E\ \texttt{to}\ e \mid \texttt{return}\ v\ \texttt{to}\ E\end{array}$$

These contexts give the core aspect calculus a call-by-value, left-to-right evaluation order, but that choice is orthogonal to the design of the language. The only requirement is that evaluation be allowed to proceed under labeled points: $l\langle E\rangle$ should be an evaluation context. This requirement ensures that the evaluation contexts accurately describe the nesting of labels as they appear in the call stack.

The operational semantics must keep track of both the labels that have been generated by the `new` construct and the advice that has been installed into the run-time environment by the program. An allocation-style semantics [12] keeps track of a set L of labels (and their associated types). Similarly, A is an ordered list of installed advice—the $\ll$ and $\gg$ operators respectively add advice to the head (left) and tail of this list. Finally, the abstract machine states or configurations C used in our operational semantics are triples, $\langle L, A, e\rangle$.

$$L ::= \cdot \mid L, l\texttt{:}t \qquad A ::= \cdot \mid A, a \qquad C ::= \langle L, A, e\rangle$$

Because the `return` operation needs to pass control to the nearest enclosing labeled point, it is convenient to define a function stack(E) that takes an evaluation context E and returns the stack of labels appearing in the context. Such stacks s, are given by the following grammar:

$$s ::= \cdot \mid l \mid s_1 :: s_2$$

The top of the stack is to the left of the list. Stack concatenation, written $s_1 :: s_2$, is associative with unit $\cdot$. The

function stack(E) is inductively defined on the structure of E, where the only interesting cases are:

$$\text{stack}([\,]) = \cdot \qquad \text{stack}(l\langle E\rangle) = \text{stack}(E) :: l$$

For the other evaluation context forms, stack(E) simply returns the recursive application of stack($-$) to the unique subcontext: stack(E << e) = stack(E), etc. As an example,

$$\text{stack}(l_1\langle(\lambda x\!:\!t.\, l_3\langle e\rangle)\; l_2\langle[\,]\rangle\rangle) = \cdot :: l_2 :: l_1$$

The operational semantics of the core aspect calculus is a transition relation $\langle L, A, e\rangle \longmapsto \langle L', A', e'\rangle$ between machine configurations consisting of the set of allocated labels, the list of installed advice, and the running program.

Most of the rules are straightforward. An auxiliary relation $\longmapsto_\beta$, defined below, gives the primitive β reductions for the language.

$$\begin{array}{rrl}
\langle L, A, (\lambda x\!:\!t.\, e)\; v\rangle & & \longmapsto_\beta \langle L, A, e\{v/x\}\rangle \\
\langle L, A, \texttt{if true then}\, e_1\, \texttt{else}\, e_2\rangle & & \longmapsto_\beta \langle L, A, e_1\rangle \\
\langle L, A, \texttt{if false then}\, e_1\, \texttt{else}\, e_2\rangle & & \longmapsto_\beta \langle L, A, e_2\rangle \\
\langle L, A, \texttt{let}\, (\vec{x}) = (\vec{v})\, \texttt{in}\, e\rangle & & \longmapsto_\beta \langle L, A, e\{\vec{v}/\vec{x}\}\rangle \\
(l \notin L) \quad \langle L, A, \texttt{new}\; x\!:\!t.\, e\rangle & & \longmapsto_\beta \langle (L, l\!:\!t), A, e\{l/x\}\rangle \\
\langle L, A, a \texttt{ << } e\rangle & & \longmapsto_\beta \langle L, (a, A), e\rangle \\
\langle L, A, a \texttt{ >> } e\rangle & & \longmapsto_\beta \langle L, (A, a), e\rangle
\end{array}$$

The first four rules are the usual β-rules for the lambda calculus with Booleans and tuples, where $e\{v/x\}$ is capture-avoiding substitution of the value v for the variable x in the expression e. The fifth rule allocates a fresh label l and substitutes it for the variable x in the scope of the `new` operator. The last two rules simply add the advice a to the appropriate end of the list. Advice at the head of the list will be run before advice at the tail.

The β-reductions apply in any evaluation context, as expressed by the following rule:

$$\frac{\langle L, A, e\rangle \longmapsto_\beta \langle L', A', e'\rangle}{\langle L, A, E[e]\rangle \longmapsto \langle L', A', E[e']\rangle}$$

The remaining constructs, advice invocation and the `return` expression, require more complex evaluation semantics.

Because multiple pieces of advice may be triggered at a single point, the operational semantics must compose them together in the order indicated by the list A. To do so, the advice $\{p.x \to e\}$ is treated as a function $\lambda x : t.\, e$, which can be combined with other advice using standard function composition. The composition is well defined because advice that accepts input of type t must produce an output of type t (or `return` to a point lower in the stack).

This behavior is captured by two auxiliary definitions. The first, $\mathcal{A}[\![A]\!]_C = e'$, takes a list of advice A and returns a function e' that is the composition of the applicable advice in the state C. The second judgment has the form $C \models p$ and is valid if the point-cut p is satisfied by the configuration C. In general, the satisfaction relation may be an arbitrary predicate on the current state of the abstract machine; Section 4 details some more point-cuts. However, in this core language, the satisfaction relation is simply defined to be the equality relation between p and the label at the current program point. The advice composition and point-cut satisfaction are defined by the following rules.

$$\frac{}{\mathcal{A}[\![\cdot]\!]_{\langle L, A, E[l\langle v\rangle]\rangle} = \lambda x\!:\!L(l).\, x}$$

$$\frac{C \models v \quad \mathcal{A}[\![A]\!]_C = \lambda y\!:\!t.\, e'}{\mathcal{A}[\![\{v.x \to e\}, A]\!]_C = \lambda x\!:\!t.\, ((\lambda y\!:\!t.\, e')\; e)}$$

$$\frac{C \not\models v \quad \mathcal{A}[\![A]\!]_C = e'}{\mathcal{A}[\![\{v.x \to e\}, A]\!]_C = e'} \qquad \frac{l = p}{\langle L, A, E[l\langle v\rangle]\rangle \models p}$$

With these definitions, the evaluation rule for $l\langle v\rangle$ simply applies the function resulting from interpreting the advice list to the value v.

$$\frac{\mathcal{A}[\![A]\!]_{\langle L, A, E[l\langle v\rangle]\rangle} = e}{\langle L, A, E[l\langle v\rangle]\rangle \longmapsto \langle L, A, E[e\; v]\rangle}$$

The expression `return` v `to` l immediately hands the value v to the nearest enclosing program point labeled by l. Using evaluation contexts and the stack($-$) function, this behavior is expressed by the following rule:

$$(l \notin \text{stack}(E)) \quad \langle L, A, l\langle E[\texttt{return}\; v\; \texttt{to}\; l]\rangle\rangle \longmapsto_\beta \langle L, A, l\langle v\rangle\rangle$$

Here, the program consists of a `return` expression in a context E labeled by l. Because the stack of labels in E does not contain the label l, the point labeled by l must be the closest such point to the `return` expression. The program thus steps immediately to the point labeled l, discarding the context E. This semantics is essentially the same as those used for exception handlers. Note that if there is no point labeled l in the context of the `return` this rule does not apply and the program will get stuck.

2.2 Type System

The type system for the core aspect calculus is a very simple extension of the type system for the base language (in this case, the simply typed lambda calculus). The main consideration is that because it is necessary to pass data back and forth between the join point of interest and the advice, the advice and control flow points must be in agreement with respect to the type of data that will exchanged. The three new types are t `label`, the type of labels that can annotate program contexts of type t, t `pc`, the type of point cuts matching program contexts of type t, and `advice`, the type of advice. Types and typing contexts are given by the following grammar:

$$\begin{array}{rcl}
t & ::= & \texttt{bool} \mid (t_1, \ldots, t_n)^{(n \geq 0)} \mid t_1 \to t_2 \\
 & \mid & t\; \texttt{label} \mid t\; \texttt{pc} \mid \texttt{advice} \\
\Gamma & ::= & \cdot \mid \Gamma, x\!:\!t
\end{array}$$

Figure 1 contains the typing rules for the new aspect expressions. These rules make use of the standard judgments of the form $\Gamma \vdash e : t$, indicating that term e can be given type t in context Γ. Boolean and tuple typing (not shown) are standard. Tuple expressions are typed by a vector of types $\vec{t}$, where $\cdot$ is the empty tuple of types (i.e. unit, inhabited by `()`), and if $\vec{t}$ and $\vec{t'}$ are tuple types, then $\vec{t}, \vec{t'}$ is their concatenation.

For simplicity, the type system is parameterized by a map L from labels to the types of the expressions they may mark. A concrete label value l is given the type t `label` whenever $L(l) = t$. The `new` expression simply introduces a new variable of type t `label`. An expression of type t `label` may be used to label another expression of type t. Since point

$$\frac{L(l) = t}{\Gamma \vdash l : t\ \texttt{label}} \qquad \frac{\Gamma, x{:}t\ \texttt{label} \vdash e : t'}{\Gamma \vdash \texttt{new}\ x{:}t.\ e : t'}$$

$$\frac{\Gamma \vdash e_1 : t\ \texttt{label} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1\langle e_2\rangle : t} \qquad \frac{\Gamma \vdash e : t\ \texttt{label}}{\Gamma \vdash e : t\ \texttt{pc}}$$

$$\frac{\Gamma \vdash e_1 : t\ \texttt{pc} \quad \Gamma, x{:}t \vdash e_2 : t}{\Gamma \vdash \{e_1.x \to e_2\} : \texttt{advice}} \qquad \frac{\Gamma \vdash e_1 : \texttt{advice} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 \ll e_2 : t}$$

$$\frac{\Gamma \vdash e_1 : \texttt{advice} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 \gg e_2 : t} \qquad \frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t\ \texttt{label}}{\Gamma \vdash \texttt{return}\ e_1\ \texttt{to}\ e_2 : t'}$$

Figure 1: Type system

types	t	::=	$\texttt{bool} \mid t_1 \to t_2$
terms	e	::=	$x \mid b \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$
		$\mid$	$\texttt{let}\ ds\ \texttt{in}\ e \mid e_1\ e_2$
decls	ds	::=	$\cdot$
		$\mid$	$(\texttt{bool}\ x = e)\ ds$
		$\mid$	$(\texttt{fun}\ f(x{:}t_1){:}t_2 = e)\ ds$
		$\mid$	$ad\ ds$
prog pts	p	::=	f
aspects	ad	::=	$\texttt{before}\ p(x) = e$
		$\mid$	$\texttt{after}\ p(x) = e$
		$\mid$	$\texttt{around}\ p(x) = e$
		$\mid$	$\texttt{around}\ p(x) = e_1; \texttt{proceed}\ y \to e_2$

Figure 2: MinAML Syntax

cuts are simply labels here, the type t `pc` is implemented by t `label`: Any expression with type t `label` may be considered to have type t `pc`.

Advice associated with a point cut of type t `pc` is constructed from code that expects a variable of type t. The body of advice must produce a result suitable for returning to the point from which the advice was triggered. Thus, the body of the advice must itself be of type t. Note that because all advice associated with a point cut p accept and produce values of the same type, it is possible to compose them in any order—the soundness of the composition used in the operational semantics follows from this constraint.

The rules for installing advice permit the program to be executed in the presence of the advice to have any type.

Lastly, the value returned to a label marking a context of type t should itself have type t. However, as with exception or continuation invocation, the `return` expression itself may be used in any context.

These rules lead to a straightforward soundness proof in the style of Wright and Felleisen [17]. A **finished** configuration is one that is either of the form $\langle L, A, v\rangle$ or of the form $\langle L, A, E[\texttt{return}\ v\ \texttt{to}\ l]\rangle$ where $l \notin \text{stack}(E)$.

A configuration $\langle L, A, e\rangle$ is well typed if, for all advice $a \in A$ it is the case that $\cdot \vdash a : \texttt{advice}$ and $\cdot \vdash e : t$ for some t (where L is the label–type map that parameterizes these type checking judgments). Given these definitions, the standard lemmas can easily be proved.

THEOREM 2.1 (PROGRESS).
If C is well typed then either the configuration is finished, or there exists another configuration C' such that $C \longmapsto C'$.

THEOREM 2.2 (PRESERVATION).
If $\langle L, A, e\rangle$ is well typed and $\langle L, A, e\rangle \longmapsto \langle L', A', e'\rangle$ then L' extends L and $\langle L', A', e'\rangle$ is well typed.

3. MinAML

This section gives a semantics for a concrete AOPL called MinAML by translating it into the core aspect calculus.

Figure 2 displays the MinAML syntax. The base types are Booleans and functions. Booleans are as usual. Function declarations define a (non-recursive) value and also implicitly declare a program point f that can be referred to by advice. Otherwise, functions are treated normally.

MinAML allows programmers to define static, second-class advice—unlike in the more general core language, programs may not manipulate advice at run-time in any significant way. Advice is immediately appended to the advice store when it is declared. In this respect, MinAML is quite similar to AspectJ.

Also like AspectJ, MinAML has three sorts of aspects: those that give advice *before* execution of point cut p (for now, p is limited to be a function call), those that give advice *after* execution of p, and those that give advice *around* p. In the first and third cases, the bound variable x will be replaced by the argument of p when the advice is triggered. In the second case, x will be replaced by p's result. When declaring around advice, the programmer can choose either to replace p entirely or to perform some pre-computation, *proceed* with p and then perform some post-computation.[2] In the latter case, after proceeding with p, a fresh variable y is bound to the result of the function.

Unlike AspectJ, which allows programmers to refer to any method that appears anywhere in their program, even private methods of classes, the functions referred to by MinAML advice must be in scope. This decision allows programmers to retain some control over basic information hiding and modularity principles in the presence of aspects. For instance, a programmer can declare a nested utility function and be assured that no advice interferes with its execution. The programmer can also decide to expose the function declaration to manipulation by advice by declaring it in an outer scope. The decision to make the external language well scoped truly is an *external language* design decision: we believe the core aspect calculus is rich enough to express AspectJ-style, scopeless advice by using a slightly different translation strategy.[3]

3.1 MinAML Interpretation

We give a semantics to well-typed MinAML programs by defining a type-directed translation into the core language.

The translation is defined by mutually recursive judg-

[2] It is straightforward to permit the proceed command to appear in arbitrary expressions inside advice, but doing so needlessly complicates the presentation without adding any further insight.

[3] Allowing programmers to reference variables defined in inner scopes would pose some (again, external language) difficulties as any simple scheme would be incompatible with the basic principles of alpha-conversion. However, these difficulties could likely be overcome by giving bindings both an internal and external name, as in Harper and Lillibridge's translucent sum calculus [7]. Once naming conventions for the external language have been overcome, the translation to internal language should be straightforward.

$$\frac{x:t \in \Gamma}{P;\Gamma \vdash x : t \overset{\text{term}}{\Longrightarrow} x} \qquad \frac{}{P;\Gamma \vdash b : \texttt{bool} \overset{\text{term}}{\Longrightarrow} b}$$

$$\frac{\begin{array}{c}P;\Gamma \vdash e_1 : \texttt{bool} \overset{\text{term}}{\Longrightarrow} e_1' \\ P;\Gamma \vdash e_2 : t \overset{\text{term}}{\Longrightarrow} e_2' \qquad P;\Gamma \vdash e_3 : t \overset{\text{term}}{\Longrightarrow} e_3'\end{array}}{P;\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : t \overset{\text{term}}{\Longrightarrow} \texttt{if}\ e_1'\ \texttt{then}\ e_2'\ \texttt{else}\ e_3'}$$

$$\frac{P;\Gamma \vdash ds; e : t \overset{\text{decs}}{\Longrightarrow} e'}{P;\Gamma \vdash \texttt{let}\ ds\ \texttt{in}\ e : t \overset{\text{term}}{\Longrightarrow} e'}$$

$$\frac{P;\Gamma \vdash e_1 : t_1 \to t_2 \overset{\text{term}}{\Longrightarrow} e_1' \quad P;\Gamma \vdash e_2 : t_1 \overset{\text{term}}{\Longrightarrow} e_2'}{P;\Gamma \vdash e_1\ e_2 : t_2 \overset{\text{term}}{\Longrightarrow} e_1'\ e_2'}$$

Figure 3: MinAML Interpretation: Terms

ments for terms, for declarations and for advice. The term translation judgment has the form $P;\Gamma \vdash e : t \overset{\text{term}}{\Longrightarrow} e'$. It computes the type t of the term e and, if it is well-formed, produces a core language term e' of the same type. The type-checking context is split into two parts. The context Γ is a mapping from MinAML variables to types. The context P is a mapping from program points p to pairs of input and output types for that program point. For example, a function $f : \texttt{bool} \to \texttt{int}$ extends the context P with the binding $f : (\texttt{bool}, \texttt{int})$ and extends the typing context Γ with $f : \texttt{bool} \to \texttt{int}$.

The term translation type checks external language terms and translates them into analogous core language constructs. All of the interesting action happens when translating declarations and advice. Figures 3, 4 and 5 present the details.

The main idea in the translation of function declarations has already been explained by example. Two new program points are declared in the course of the translation, one for the function entry point (f_{pre}) and one for the exit point (f_{post}). These two points may be used in advice definitions declared in the following scope. The translation maintains the invariant that if the binding $p : (t_1, t_2)$ appears in P then the translated term will type check in a context extended with $p_{\text{pre}} : t_1\ \texttt{label}, p_{\text{post}} : t_2\ \texttt{label}$.

The main ideas for the aspect translation have also been explained informally in previous sections. *Before* advice for p is defined to be core language advice triggered by the p_{pre} join point. *After* advice for p is triggered by the p_{post} join point. *Around* advice with a proceed statement defines two pieces of advice, one for the p_{pre} point and one for the p_{post} point. Finally, *around* advice without a proceed statement is triggered by p_{pre} but returns to p_{post}.

The main property of the translation is that it produces well-typed core language terms. Define $\mathcal{P}(p : (t_1, t_2))$ to be the context $p_{\text{pre}} : t_1\ \texttt{label}, p_{\text{post}} : t_2\ \texttt{label}$ and let $\mathcal{P}(P)$ be the point-wise extension of the former translation.

LEMMA 3.1 (TRANSLATION TYPE PRESERVATION).

1. *If* $P;\Gamma \vdash e : t \overset{\text{term}}{\Longrightarrow} e'$ *then* $\Gamma, \mathcal{P}(P) \vdash e' : t$.
2. *If* $P;\Gamma \vdash ds; e : t \overset{\text{decs}}{\Longrightarrow} e'$ *then* $\Gamma, \mathcal{P}(P) \vdash e' : t$.
3. *If* $P;\Gamma \vdash ad \overset{\text{adv}}{\Longrightarrow} e'$ *then* $\Gamma, \mathcal{P}(P) \vdash e' : \texttt{advice}$.

$$\frac{P;\Gamma \vdash e : t \overset{\text{term}}{\Longrightarrow} e'}{P;\Gamma \vdash \cdot; e : t \overset{\text{decs}}{\Longrightarrow} e'}$$

$$\frac{P;\Gamma \vdash e_1 : \texttt{bool} \overset{\text{term}}{\Longrightarrow} e_1' \quad P;\Gamma, x{:}\texttt{bool} \vdash ds; e_2 : t \overset{\text{decs}}{\Longrightarrow} e_2'}{P;\Gamma \vdash (\texttt{bool}\ x = e_1)\ ds; e_2 : t \overset{\text{decs}}{\Longrightarrow} \texttt{let}\ x{:}\texttt{bool} = e_1'\ \texttt{in}\ e_2'}$$

$$\frac{\begin{array}{c}P;\Gamma, x{:}t_1 \vdash e_1 : t_2 \overset{\text{term}}{\Longrightarrow} e_1' \\ P, f : (t_1, t_2); \Gamma, f : t_1 \to t_2 \vdash ds; e_2 : t \overset{\text{decs}}{\Longrightarrow} e_2'\end{array}}{\begin{array}{l}P;\Gamma \vdash (\texttt{fun}\ f\ (x : t_1) : t_2 = e_1)\ ds; e_2 : t \overset{\text{decs}}{\Longrightarrow} \\ \quad \texttt{new}\ f_{\text{pre}} : t_1.\ \texttt{new}\ f_{\text{post}} : t_2.\ \texttt{let}\ f = e_b\ \texttt{in}\ e_2'\end{array}}$$

where $e_b = \lambda x : t_1.f_{\text{post}}\langle \texttt{let}\ x : t_1 = f_{\text{pre}}\langle x \rangle\ \texttt{in}\ e_1' \rangle$

$$\frac{P;\Gamma \vdash ad \overset{\text{adv}}{\Longrightarrow} e_1' \quad P;\Gamma \vdash ds; e_2 : t \overset{\text{decs}}{\Longrightarrow} e_2'}{P;\Gamma \vdash ad\ ds; e_2 : t \overset{\text{decs}}{\Longrightarrow} e_1' \gg e_2'}$$

Figure 4: MinAML Interpretation: Declarations

$$\frac{p : (t_1, t_2) \in P \quad P;\Gamma, x : t_1 \vdash e : t_1 \overset{\text{term}}{\Longrightarrow} e'}{P;\Gamma \vdash \texttt{before}\ p(x) = e \overset{\text{adv}}{\Longrightarrow} \{p_{\text{pre}}.x \to e'\}}$$

$$\frac{p : (t_1, t_2) \in P \quad P;\Gamma, x : t_2 \vdash e : t_2 \overset{\text{term}}{\Longrightarrow} e'}{P;\Gamma \vdash \texttt{after}\ p(x) = e \overset{\text{adv}}{\Longrightarrow} \{p_{\text{post}}.x \to e'\}}$$

$$\frac{\begin{array}{c}p : (t_1, t_2) \in P \\ P;\Gamma, x : t_1 \vdash e_1 : t_1 \overset{\text{term}}{\Longrightarrow} e_1' \qquad P;\Gamma, y : t_2 \vdash e_2 : t_2 \overset{\text{term}}{\Longrightarrow} e_2'\end{array}}{\begin{array}{l}P;\Gamma \vdash \texttt{around}\ p(x) = e_1; \texttt{proceed}\ y \to e_2 \overset{\text{adv}}{\Longrightarrow} \\ \quad \{p_{\text{pre}}.x \to e_1'\} \gg \{p_{\text{post}}.y \to e_2'\}\end{array}}$$

$$\frac{p : (t_1, t_2) \in P \quad P;\Gamma, x : t_1 \vdash e : t_2 \overset{\text{term}}{\Longrightarrow} e'}{\begin{array}{l}P;\Gamma \vdash \texttt{around}\ p(x) = e \overset{\text{adv}}{\Longrightarrow} \\ \quad \{p_{\text{pre}}.x \to \texttt{return}\ e'\ \texttt{to}\ p_{\text{post}}\}\end{array}}$$

Figure 5: MinAML Interpretation: Aspects

The proof of Lemma 3.1 is by induction on the translation derivation. Combining Lemma 3.1 with the type safety result for the core language yields an important safety result for MinAML.

THEOREM 3.1 (MINAML SAFETY).
Suppose that $\cdot;\cdot \vdash e : t \overset{\text{term}}{\Longrightarrow} e'$. *Then either* e' *fails to terminate or there is a finished configuration* $\langle L, A, e'' \rangle$ *such that* $\langle \cdot, \cdot, e' \rangle \longmapsto^\star \langle L, A, e'' \rangle$

3.2 Objects

The bulk of this paper focuses on using aspects in the context of a purely functional language. However, we have tried to design the core language so that each feature is *orthogonal* to the others. In particular, the labeled join points are defined independently of other constructs and hence can be reused in other computational settings with little change. In order to justify this claim, we have lifted Abadi and Cardelli's first-order object calculus (AC) directly

from their textbook [1]. This section shows how the aspect language constructs interoperate with it. The main point is that while we naturally need to add objects to both the external and core languages, the semantics of join points remains unchanged. Moreover, while additional syntax is needed in the external language to allow programmers to refer to new join points, the underlying semantics of advice also remains the same. This analysis provides evidence that the semantic framework is both general and robust.

3.2.1 Object-oriented Core Language

The type system and syntax for the AC object-oriented language is taken directly from Abadi and Cardelli [1].

$$\begin{array}{lcl} t & ::= & \cdots \mid [m_i{:}t_i]^{1..n} \\ e & ::= & \cdots \mid [m_i = \varsigma\, x_i.e_i]^{1..n} \mid e.m \mid e_1.m \Leftarrow \varsigma\, x.e_2 \\ v & ::= & \cdots \mid [m_i = \varsigma\, m_i.e_i]^{1..n} \end{array}$$

AC is a classless language. New objects $[m_i = \varsigma\, x_i.e_i]^{1..n}$ may be defined at any point in a computation. The superscript $1..n$ indicates there is a series of n method declarations in the object. Method invocation is denoted $e.m$ and method update (override) is denoted $e_1.m \Leftarrow \varsigma\, x.e_2$.

The AC typing rules are straightforward. We have modified them to permit labels, and slightly more significantly, we have dropped the subtyping for the sake of simplicity.

$$\frac{\Gamma, x{:}[m_i{:}t_i]^{1..n} \vdash e_i : t_i}{\Gamma \vdash [m_i = \varsigma\, x_i.e_i]^{1..n} : [m_i{:}t_i]^{1..n}}$$

$$\frac{\Gamma \vdash e : [m_i{:}t_i]^{1..n} \quad 1 \le j \le n}{\Gamma \vdash e.m_j : t_j}$$

$$\frac{\Gamma \vdash e_1 : [m_i{:}t_i]^{1..n} \quad \Gamma, x{:}[m_i{:}t_i]^{1..n} \vdash e_2 : t_j \quad 1 \le j \le n}{\Gamma \vdash e_1.m_j \Leftarrow \varsigma\, x.e_2 : [m_i{:}t_i]^{1..n}}$$

Finally, to extend the operational semantics, we define further evaluation contexts corresponding to the new expression forms and the appropriate beta rules.

Evaluation Contexts:

$$E \quad ::= \quad \cdots \mid E.m \mid E.m \Leftarrow \varsigma\, x.e_2$$

Beta Rules:

$$\begin{array}{l} \langle L, A, [m_i = \varsigma\, x_i.e_i]^{1..n}.m_j \rangle \longmapsto_\beta \\ \quad \langle L, A, e_j\{[m_i = \varsigma\, x_i.e_i]^{1..n}/x_j\} \rangle \\ \langle L, A, [m_i = \varsigma\, x_i.e_i]^{1..n}.m_j \Leftarrow \varsigma\, x.e \rangle \longmapsto_\beta \\ \quad \langle L, A, [m_1 = \varsigma\, x_1.e_1, \ldots, m_j = \varsigma\, x.e, \ldots, m_n = \varsigma\, x_n.e_n] \rangle \end{array}$$

To adapt the progress and preservation theorems stated in the previous section, we need only fill in the inductive cases for objects; the overall proof structure remains intact.

3.2.2 Object-oriented External Language

The external language requires a new type for objects, new declarations for defining objects and new expression forms for method invocation and update. In addition, we add an expression form to control monitoring of method updates. The declaration `monitor` $t.m$ specifies that any update of method m to an object with type t may be intercepted and modified by advice. This declaration also introduces a new join point $t.m$, and programmers can declare before, after and around advice that will be triggered by that join point (i.e., triggered whenever the associated method update occurs). Programmers can also declare advice triggered by calls to the m method of object x via the join point $x.m$.

$$\begin{array}{lcl} t & ::= & \cdots \mid [m_i{:}t_i]^{1..n} \\ e & ::= & \cdots \mid e.m \mid e_1.m \Leftarrow \varsigma\, x.e_2 \\ d & ::= & \cdots \mid (\texttt{object}\, x{:}t = [m_i = \varsigma\, x_i.e_i]^{1..n})\ ds \\ & \mid & \texttt{monitor}\, t.m\ ds \\ p & ::= & \cdots \mid x.m \mid t.m \end{array}$$

As a simple example, consider the following code which declares an object with two fields. One field holds an integer and the other holds a function that adds the integer to its argument. To prevent the integer field from being updated (effectively rendering it "const"), the program declares that the field is monitored and installs around advice that replaces any attempted update with the identity function.

```
let object x:t =
      [i    = ςs.3;
       plus = ςs.let fun f x = s.i + x in f]
     monitor t.i
     around (t.i) (x) = x
in ...
where t = [i : int; plus : int -> int]
```

Interpreting the object-oriented source language in the core aspect calculus poses no challenges. The `monitor` declaration translates to a pair of expressions that allocate new pre- and post-labels used to mark method updates. Interpreting both method update in the case that the update is monitored, and object declarations, follows a similar strategy to compilation of function bodies. The translation marks the control-flow points just prior to and just after the operation in question. Advice declarations in the same scope can manipulate these program points just as they manipulate function entry and exit points. The full details have been omitted due to space considerations.

4. Complex Point Cuts

This section investigates two further generalizations of the basic aspect framework. The first generalization allows advice to be associated with a set of labels instead of just one label, which permits the code of the advice to be shared by many program points. The second generalization is to permit run-time inspection of the labels that appear in the call stack, which allows advice to make context sensitive decisions about how to modify the program.

4.1 Label Sets

The first generalization associates a set of labels with each piece of advice. Doing so is useful in situations where the same advice is applied at many different locations. For example, one might want to instrument a collection of related functions of type $t_1 \to t_2$ with the same preprocessing of the argument, yet still allow the possibility of associating other, different advice with each function. With sets of labels, this situation can be expressed as:

```
new pre1:t1.new pre2:t1.
{{pre1,pre2}.x→e1} >>  // Runs at either point
{{pre1}.y→e2} >>       // Runs at pre1
{{pre2}.z→e3} >>       // Runs at pre2
  let f = λx:t1. let x = pre1⟨x⟩ in ... in
  let g = λx:t1. let x = pre2⟨x⟩ in ... in ...
```

The necessary change to the syntax of the language is minimal, as shown in the grammar below:

$$\begin{array}{rcl} e & ::= & \cdots \mid \{e_1, \ldots, e_n\} \mid e_1 \cup e_2 \mid e_1 \cap e_2 \\ v & ::= & \cdots \mid \{v_1, \ldots, v_n\} \end{array}$$

The advice $\{\{l_1, \ldots, l_n\}.x \to e\}$ is triggered whenever a point labeled by any of the labels l_1 through l_n is reached.

To change the operational semantics of advice invocation, we simply replace the definition of the satisfaction relation with the following:

$$\frac{l \in \{l_1, \ldots, l_n\}}{\langle L, A, E[l\langle v\rangle]\rangle \models \{l_1, \ldots, l_n\}}$$

Advice is still applied in the order defined by the list A, but now advice is triggered by a label l if l is in the set. Evaluation semantics for the set operators $e_1 \cup e_2$ and $e_1 \cap e_2$ are straightforward to define.

The type system is altered to use the following rules for type checking point cuts. The type t `pc` is now implemented by a set of labels of the same type.

$$\frac{(\Gamma \vdash e_i : t\ \texttt{label})^{(1 \le i \le n)}}{\Gamma \vdash \{e_1, \ldots, e_n\} : t\ \texttt{pc}} \qquad \frac{\Gamma \vdash e_1 : t\ \texttt{pc} \quad \Gamma \vdash e_2 : t\ \texttt{pc}}{\Gamma \vdash e_1 \cup e_2 : t\ \texttt{pc}}$$

$$\frac{\Gamma \vdash e_1 : t\ \texttt{pc} \quad \Gamma \vdash e_2 : t\ \texttt{pc}}{\Gamma \vdash e_1 \cap e_2 : t\ \texttt{pc}}$$

One could imaging further refinements along these lines. For instance, one refinement would be to put more structure on the labels themselves, perhaps by introducing a hierarchy of labels. A piece of advice would then be triggered by its label or any label lower in the tree. Including a "top" label in the hierarchy would let one define advice that is triggered whenever any labeled point is reached.

4.1.1 MinAML Extensions and Interpretation

Extending MinAML's point cut language to include sets of labels requires some minor adjustments to the syntax:

$$\begin{array}{rcl} pc & ::= & \{p_1, \ldots, p_n\} \\ ad & ::= & \texttt{before}\ pc(x) = e \\ & \mid & \texttt{after}\ pc(x) = e \\ & \mid & \texttt{around}\ pc(x) = e \\ & \mid & \texttt{around}\ pc(x) = e_1; \texttt{proceed}\ y \to e_2 \end{array}$$

The interpretation also requires some adjustments. One problem is that around advice can be called from multiple different labeled points, so it is impossible to determine statically which label it should return to. To circumvent this difficulty, the translation uses first-class labels: the around advice is passed the "continuation" label it should return to.

The new translation of function and advice declarations appears in Figure 6. Given a set s of source-level program points $\{p_1, \ldots, p_n\}$, we use the meta-level function $\texttt{pre}(s)$ to generate the corresponding set of labels $\{p_{1,\texttt{pre}}, \ldots, p_{n,\texttt{pre}}\}$. The function $\texttt{post}(s)$ is similar. The translation of object expressions (omitted) can be dealt with analogously.

$$\frac{P; \Gamma, x : t_1 \vdash e_1 : t_2 \overset{\text{term}}{\Longrightarrow} e_1' \qquad P, f : (t_1, t_2); \Gamma, f : t_1 \to t_2 \vdash ds; e_2 : t \overset{\text{decs}}{\Longrightarrow} e_2'}{P; \Gamma \vdash (\texttt{fun}\ f\ (x : t_1) : t_2 = e_1)\ ds; e_2 : t \overset{\text{decs}}{\Longrightarrow} \texttt{new}\ f_{\texttt{pre}} : t_1 \times t_2\ \texttt{label}.\ \texttt{new}\ f_{\texttt{post}} : t_2.\ \texttt{let}\ f = e_b\ \texttt{in}\ e_2'}$$

$$e_b \overset{\text{def}}{=} \lambda x : t_1.f_{\texttt{post}}\langle \texttt{let}\ (x, \_) = f_{\texttt{pre}}\langle (x, f_{\texttt{post}})\rangle\ \texttt{in}\ e_1'\rangle$$

$$\frac{(p : (t_1, t_2) \in P)^{p \in s} \quad P; \Gamma, x : t_1 \vdash e : t_1 \overset{\text{term}}{\Longrightarrow} e'}{P; \Gamma \vdash \texttt{before}\ s(x) = e \overset{\text{adv}}{\Longrightarrow} \{\texttt{pre}(s).x \to \texttt{let}\ (x, l) = x\ \texttt{in}\ (e', l)\}}$$

$$\frac{(p : (t_1, t_2) \in P)^{p \in s} \quad P; \Gamma, x : t_2 \vdash e : t_2 \overset{\text{term}}{\Longrightarrow} e'}{P; \Gamma \vdash \texttt{after}\ s(x) = e \overset{\text{adv}}{\Longrightarrow} \{\texttt{post}(s).x \to e'\}}$$

$$\frac{(p : (t_1, t_2) \in P)^{p \in s} \quad P; \Gamma, x : t_1 \vdash e_1 : t_1 \overset{\text{term}}{\Longrightarrow} e_1' \quad P; \Gamma, y : t_2 \vdash e_2 : t_2 \overset{\text{term}}{\Longrightarrow} e_2'}{P; \Gamma \vdash \texttt{around}\ s(x) = e_1; \texttt{proceed}\ y \to e_2 \overset{\text{adv}}{\Longrightarrow} \{\texttt{pre}(s).x \to \texttt{let}\ (x, l) = x\ \texttt{in}\ (e_1', l)\}\ \texttt{>>}\ \{\texttt{post}(s).y \to e_2'\}}$$

$$\frac{(p : (t_1, t_2) \in P)^{p \in s} \quad P; \Gamma, x : t_1 \vdash e : t_2 \overset{\text{term}}{\Longrightarrow} e'}{P; \Gamma \vdash \texttt{around}\ s(x) = e \overset{\text{adv}}{\Longrightarrow} \{\texttt{pre}(s).x \to \texttt{let}\ (x, l) = x\ \texttt{in}\ \texttt{return}\ e'\ \texttt{to}\ l\}}$$

Figure 6: MinAML Interpretation: Label Sets

4.2 Stack Patterns

While labeled program expressions suffice to capture some of the "interesting" program points, whether a point is "interesting" often depends on context. For example, a typical use of aspects for debugging is to print the arguments of a function f when it is called from inside the body of a second function, g. The debugging advice is not invoked when f is called from some third function h. To enable this application, AOPLs provide mechanisms that allow the programmer to specify in what dynamic contexts advice should be triggered.

One could tie this contextual information into the advice construct itself, but it seems more general to provide an orthogonal mechanism for querying the run-time state of the program. This section proposes *stack patterns* as a way to achieve the desired expressiveness without altering the advice; this approach leads to a cleaner semantics.

During the course of evaluation, the labeled program points naturally form a stack, which is a useful model of the computation being carried out by the program. The `return` expression already makes use of this fact to determine to which point control should be passed. Stack patterns allow programmers to write queries over the label stack.

Aspect-oriented languages also permit queries on the *data* stored in the run-time stack. This facility is useful for writing *point-cut designators*: triggers that depend on context. To handle this feature, we extend the core language with a means of storing data values in the stack by adding a new expression $\texttt{store}\ x : t = e_1\ \texttt{in}\ e_2$. The semantics of `store` is like an ordinary `let` except that the substitution of the value for the bound variable is performed explicitly—evaluation proceeds within the body of the `store`. The evaluation contexts are extended to include:

$$E ::= \ldots \mid \texttt{store}\ x = E\ \texttt{in}\ e \mid \texttt{store}\ x = v\ \texttt{in}\ E$$

Two additional β-rules model the explicit substitutions; here, the function $svars(E)$ yields the set of variables x such that $E = E'[\texttt{store}\ x = v\ \texttt{in}\ E'']$.

$$\begin{array}{l}\langle L, A, \texttt{store}\ x = v\ \texttt{in}\ E[x]\rangle \longmapsto_\beta \\ \quad \langle L, A, \texttt{store}\ x = v\ \texttt{in}\ E[v]\rangle \qquad (x \notin svars(E)) \\ \langle L, A, \texttt{store}\ x = v\ \texttt{in}\ v'\rangle \longmapsto_\beta \langle L, A, v'\rangle\end{array}$$

Allowing evaluation to proceed under the `store` binding means that the stack embodied by the evaluation contexts now includes the stored data. Thus, we can extend stacks to include values ($\texttt{val}:t = v$) in addition to the labels, and extend the stack($-$) function to extract the data too:

$$\begin{array}{l} s ::= \cdot \mid l \mid s_1 :: s_2 \mid \texttt{val}:t = v \\ \text{stack}(\texttt{store}\ x:t = v\ \texttt{in}\ E) = \text{stack}(E) :: (\texttt{val}:t = v) \\ \text{stack}(\texttt{store}\ x:t = E\ \texttt{in}\ e) = \text{stack}(E)\end{array}$$

Stack patterns are expressions that describe the stack of labels and stored values present in the dynamic evaluation context—the metavariable *pat* is used to emphasize that a given expression is in fact a stack pattern. A stack pattern is similar to a regular expression over labels, but it may also bind stored values. The grammar below summarizes the additions to the base language needed for stack patterns.

$$\begin{array}{rcl} e & ::= & \ldots \mid l \mid e_1; e_2 \mid e_1 \mid e_2 \mid e^* \mid e_1\ \texttt{\&}\ e_2 \mid \neg e \\ & \mid & \texttt{val}:t \mid \texttt{match}[t](e)\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\ v & ::= & \ldots \mid l \mid v_1; v_2 \mid v_1 \mid v_2 \mid v^* \mid v_1\ \texttt{\&}\ v_2 \mid \neg v \\ & \mid & \texttt{val}:t \end{array}$$

The pattern l matches the stack consisting of the label l. Concatenation of pattern expressions is written $e_1; e_2$, union is written $e_1 \mid e_2$, and a Kleene star operator is written e^*. The intersection operator e_1 `&` e_2 matches patterns in the intersection of those matched by e_1 and e_2, binding the data from both patterns. A negation pattern, $\neg e$, matches a stack if there is no possible way to parse the stack successfully according to e. The pattern $\texttt{val}:t$ matches a value of type t stored in the stack. Booleans `true` and `false` are patterns that respectively match all stacks and no stacks.

The program form $\texttt{match}[t](pat)\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ attempts to match the pattern *pat* against the dynamic stack, extracting any data bound by $\texttt{val}:t$ patterns. If the stack successfully matches the pattern, the expression e_1, which must be a function, is applied to the extracted data. Otherwise the expression e_2 is evaluated.

Consider instrumenting a function f with pre- and post-labels as in the translation from MinAML. Using `store` rather than an ordinary `let` to bind the argument to f gives the following:

$$\lambda x{:}t.\ f_{\texttt{post}}\langle \texttt{store}\ x = f_{\texttt{pre}}\langle x\rangle\ \texttt{in}\ e\rangle$$

This new translation allows a stack pattern to extract the argument passed to f. For example, one can write a piece of advice that takes action only when g is called directly from the body of f. In the example, the f's argument is bound to the variable `y` in the expression e.

```
{g_pre.x →  match(g_pre;g_post;val:t;f_post;true)
            then λy:t. e     // take action
            else x}          // just continue
```

The stack matches the pattern $g_{\texttt{pre}}; g_{\texttt{post}}; \texttt{val}:t; f_{\texttt{post}}; \texttt{true}$ only when control is inside the precondition advice of g but before leaving the scope of f. (The tail of the stack, matched by `true`, can be anything.) There is some subtlety here, though: Unless *all* functions have been instrumented with pre- and post-labels, there might be calls to arbitrarily many unlabeled functions between the $f_{\texttt{post}}$ and $g_{\texttt{pre}}$. On the other hand, this regular expression does not permit any labels to appear between $f_{\texttt{post}}$ and $g_{\texttt{post}}$ on the stack. To allow that situation, the regular expression $g_{\texttt{pre}}; g_{\texttt{post}}; \texttt{true}; \texttt{val}:t; f_{\texttt{post}}; \texttt{true}$ could be used instead. Which label is desirable depends on the situation. The point is that this framework is flexible enough to express many possible choices, several more of which are explored in the translation of an extended variant of MinAML discussed below.

$$s \models \texttt{true} \Rightarrow \cdot \qquad l \models l \Rightarrow \cdot \qquad \texttt{val}:t = v \models \texttt{val}:t \Rightarrow v$$

$$\frac{s \models pat_1 \Rightarrow \vec{v_1} \quad s \models pat_2 \Rightarrow \vec{v_2}}{s \models pat_1\ \texttt{\&}\ pat_2 \Rightarrow \vec{v_1}, \vec{v_2}} \qquad \frac{s \models pat_1 \Rightarrow \vec{v_1}}{s \models pat_1 \mid pat_2 \Rightarrow \vec{v_1}}$$

$$\frac{s_1 \models pat_1 \Rightarrow \vec{v_1} \quad s_2 \models pat_2 \Rightarrow \vec{v_2}}{s_1 :: s_2 \models pat_1; pat_2 \Rightarrow \vec{v_1}, \vec{v_2}} \qquad \frac{s \models pat_2 \Rightarrow \vec{v_2}}{s \models pat_1 \mid pat_2 \Rightarrow \vec{v_2}}$$

$$\cdot \models pat^* \Rightarrow \cdot \qquad \frac{s_1 \models pat \Rightarrow \cdot \quad s_2 \models pat^* \Rightarrow \cdot}{s_1 :: s_2 \models pat^* \Rightarrow \cdot} \qquad \frac{s \not\models pat \Rightarrow \vec{v}}{s \models \neg pat \Rightarrow \cdot}$$

Figure 7: Stack Pattern Interpretation

Besides adding additional evaluation contexts to handle the evaluation of the stack patterns themselves, the operational semantics must define the behavior of the `match` expression. Two additional rules are needed:

$$\frac{\text{stack}(E) \models pat \Rightarrow \vec{v}}{\langle L, A, E[\texttt{match}[t](pat)\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\rangle \longmapsto \langle L, A, E[e_1\ (\vec{v})]\rangle}$$

$$\frac{\text{stack}(E) \not\models pat \Rightarrow \vec{v}}{\langle L, A, E[\texttt{match}[t](pat)\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\rangle \longmapsto \langle L, A, E[e_2]\rangle}$$

In these evaluation rules, the judgment $s \models pat \Rightarrow \vec{v}$ determines when the stack s matches the pattern *pat*. If so, any values matched by *pat* are returned in the vector $\vec{v}$. A reasonable implementation of stack matching would be to restrict the regular expressions to be second class values by permitting only matches against stack predicate values. This would permit the compiler to generate an efficient automaton for pattern matching; however, for the sake of generality, we consider the fully dynamic case here. It is easy to specify the implementation of $s \models pat \Rightarrow \vec{v}$ using the nondeterministic inference rules shown in Figure 7.

The only remaining issue is how to assign types to patterns. The simplest solution is to give stack patterns their own type, t `pat`, the type of patterns that binds data of type t. Booleans and labels can be treated as stack predicates—rather than use full-blown subtyping, the rules in Figure 8 instead permit the coercion directly. The additional type machinery is straightforward: stack patterns may be built compositionally out of stack patterns, and the `match` expression takes a pattern, function to handle the successful match, and an expression to return in case of match failure.

4.2.1 MinAML Extensions and Interpretation

Extending MinAML with richer *point-cut designators* in the style of AspectJ requires a change to the source syntax, as shown in the following grammar.

$$\frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash e : \cdot\ \texttt{pat}} \quad \frac{\Gamma \vdash e : t\ \texttt{label}}{\Gamma \vdash e : \cdot\ \texttt{pat}} \quad \frac{\Gamma \vdash e : \vec{t}\ \texttt{pat}}{\Gamma \vdash \neg e : \cdot\ \texttt{pat}}$$

$$\frac{\Gamma \vdash e_1 : \vec{t_1}\ \texttt{pat} \quad \Gamma \vdash e_2 : \vec{t_2}\ \texttt{pat}}{\Gamma \vdash e_1\ \texttt{\&}\ e_2 : \vec{t_1}, \vec{t_2}\ \texttt{pat}} \quad \frac{\Gamma \vdash e : \cdot\ \texttt{pat}}{\Gamma \vdash e^* : \cdot\ \texttt{pat}}$$

$$\frac{\Gamma \vdash e_1 : \vec{t}\ \texttt{pat} \quad \Gamma \vdash e_2 : \vec{t}\ \texttt{pat}}{\Gamma \vdash e_1 \mid e_2 : \vec{t}\ \texttt{pat}} \quad \frac{}{\Gamma \vdash \texttt{val}\,t : t\ \texttt{pat}}$$

$$\frac{\Gamma \vdash e_1 : \vec{t_1}\ \texttt{pat} \quad \Gamma \vdash e_2 : \vec{t_2}\ \texttt{pat}}{\Gamma \vdash e_1; e_2 : \vec{t_1}, \vec{t_2}\ \texttt{pat}} \quad \frac{\Gamma \vdash e : \cdot\ \texttt{pat}}{\Gamma \vdash e^* : \cdot\ \texttt{pat}}$$

$$\frac{\Gamma \vdash e : \vec{t}\ \texttt{pat} \quad \Gamma \vdash e_1 : \vec{t} \to t' \quad \Gamma \vdash e_2 : t'}{\Gamma \vdash \texttt{match}[t](e)\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t'}$$

Figure 8: Stack Pattern Typing

$$\frac{(f : (t_1, t_2) \in P)}{P; \Gamma \vdash \texttt{within}f(x) : x : t_1 \overset{\texttt{pcd}}{\Longrightarrow} (\texttt{val}\,t_1); f_{\texttt{post}}; \texttt{true}}$$

$$\frac{P; \Gamma \vdash pcd : \Gamma' \overset{\texttt{pcd}}{\Longrightarrow} pat}{P; \Gamma \vdash \texttt{cflow}(pcd) : \Gamma' \overset{\texttt{pcd}}{\Longrightarrow} \neg(\texttt{true}; pat; \texttt{true}); pat; \texttt{true}}$$

$$\frac{P; \Gamma \vdash pcd : \Gamma' \overset{\texttt{pcd}}{\Longrightarrow} pat}{P; \Gamma \vdash \texttt{cflowtop}(pcd) : \Gamma' \overset{\texttt{pcd}}{\Longrightarrow} \texttt{true}; pat; \neg(\texttt{true}; pat; \texttt{true})}$$

Figure 9: MinAML Interpretation: Point-cut designators

$$\begin{array}{rcl} pcd & ::= & \texttt{within}f(x) \mid pcd_1\ \texttt{\&}\ pcd_2 \mid pcd_1 \mid pcd_2 \\ & \mid & \neg pcd \mid \texttt{cflow}(pcd) \mid \texttt{cflowtop}(pcd) \\ ad & ::= & \texttt{before}\ pc(x)\ \texttt{when}\ pcd = e \\ & \mid & \texttt{after}\ pc(x)\ \texttt{when}\ pcd = e \\ & \mid & \texttt{around}\ pc(x)\ \texttt{when}\ pcd = e \\ & \mid & \texttt{around}\ pc(x)\ \texttt{when}\ pcd = e_1; \texttt{proceed}\,y \to e_2 \end{array}$$

The `when` clauses specify that the advice will be triggered only under the conditions given by the point-cut designator pcd. The $\texttt{within}f(x)$ designator says that the advice is triggered only when control is immediately inside the body of the function f (with no intervening calls). In this case, the variable x is bound to the argument passed to f. The $\texttt{cflow}(pcd)$ designator says that the advice is triggered if a part of the calling context matches pcd, where the arguments bound in the pcd are the *nearest* such call . For example, $\texttt{cflow}(\texttt{within}f(x))$ means that the advice may be triggered from within arbitrarily deeply nested function calls reachable from within the body of f and that x will be bound to the most recent arguments passed to f. Designator $\texttt{cflowtop}(pcd)$ is similar to `cflow` except that the arguments are bound to the earliest calls rather than the most recent. The pcd_1 `&` pcd_2 designator requires the context to match both pcd_1 and pcd_2, and $pcd_1 \mid pcd_2$ requires the context to match at least one of pcd_1 and pcd_2. Negation, $\neg pcd$, holds if there is no possible way of parsing the stack to match pcd.

The new translation assumes that function arguments are `store`-bound rather than `let`-bound. With that slight change to the MinAML translation, the stack, except for the top of the stack, is guaranteed to look like:

$$\texttt{val}\,t^n :: f^n_{\texttt{post}} :: \texttt{val}\,t^{n-1} :: f^{n-1}_{\texttt{post}} :: \ldots :: \texttt{val}\,t^0 :: f^0_{\texttt{post}} \quad (\star)$$

The top of the stack is either $f^{n+1}_{\texttt{pre}}$; $f^{n+1}_{\texttt{post}}$ or $f^{n+1}_{\texttt{post}}$, depending on whether execution is just entering or just leaving f.

A point-cut designator translates to a stack pattern. The three interesting cases are shown in Figure 9 (the remaining cases are straightforward). The patterns assume that the stack is of the form $(\star)$—the translation of the advice declarations themselves take care of the rest of the pattern, as shown in Figure 10. This translation handles label sets as well as `when` clauses, so it threads the `return` label through the aspects.

$$\frac{\begin{array}{c}(p : (t_1, t_2) \in P)^{p \in s} \qquad P; \Gamma \vdash pcd : \Gamma' \overset{\texttt{pcd}}{\Longrightarrow} pat \\ P; \Gamma, \Gamma', x : t_1 \vdash e : t_1 \overset{\texttt{term}}{\Longrightarrow} e' \qquad \Gamma' \overset{\texttt{ctx}}{\Longrightarrow} t\end{array}}{P; \Gamma \vdash \texttt{before}\ s(x)\ \texttt{when}\ pcd = e \overset{\texttt{adv}}{\Longrightarrow} Pre(s, x, t, pat, \Gamma', e')}$$

$$\frac{\begin{array}{c}(p : (t_1, t_2) \in P)^{p \in s} \qquad P; \Gamma \vdash pcd : \Gamma' \overset{\texttt{pcd}}{\Longrightarrow} pat \\ P; \Gamma, \Gamma', x : t_2 \vdash e : t_2 \overset{\texttt{term}}{\Longrightarrow} e' \qquad \Gamma' \overset{\texttt{ctx}}{\Longrightarrow} t\end{array}}{P; \Gamma \vdash \texttt{after}\ s(x)\ \texttt{when}\ pcd = e \overset{\texttt{adv}}{\Longrightarrow} Post(s, x, t, pat, \Gamma', e')}$$

$$\frac{\begin{array}{c}(p : (t_1, t_2) \in P)^{p \in s} \quad P; \Gamma \vdash pcd : \Gamma' \overset{\texttt{pcd}}{\Longrightarrow} pat \quad \Gamma' \overset{\texttt{ctx}}{\Longrightarrow} t \\ P; \Gamma, \Gamma', x : t_1 \vdash e_1 : t_1 \overset{\texttt{term}}{\Longrightarrow} e'_1 \quad P; \Gamma, \Gamma', y : t_2 \vdash e_2 : t_2 \overset{\texttt{term}}{\Longrightarrow} e'_2\end{array}}{\begin{array}{l}P; \Gamma \vdash \texttt{around}\ s(x)\ \texttt{when}\ pcd = e_1; \texttt{proceed}\,y \to e_2 \overset{\texttt{adv}}{\Longrightarrow} \\ Pre(s, x, t, pat, \Gamma, e'_1)\ \texttt{>>}\ Post(s, x, t, pat, \Gamma, e'_2)\end{array}}$$

$$\frac{\begin{array}{c}(p : (t_1, t_2) \in P)^{p \in s} \qquad P; \Gamma \vdash pcd : \Gamma' \overset{\texttt{pcd}}{\Longrightarrow} pat \\ \Gamma' \overset{\texttt{ctx}}{\Longrightarrow} t \qquad P; \Gamma, \Gamma', x : t_1 \vdash e : t_2 \overset{\texttt{term}}{\Longrightarrow} e'\end{array}}{\begin{array}{c}P; \Gamma \vdash \texttt{around}\ s(x)\ \texttt{when}\ pcd = e \overset{\texttt{adv}}{\Longrightarrow} \\ Return(s, x, t, pat, \Gamma, e')\end{array}}$$

$$\begin{array}{l} Pre(s, x, t, pat, \Gamma, e) \overset{\text{def}}{=} \\ \{\texttt{pre}(s).x \to\ \texttt{let}\ (x, l) = x\ \texttt{in}\ PreBody(s, x, t, pat, \Gamma, e) \end{array}$$

$$\begin{array}{l} Return(s, x, t, pat, \Gamma, e) \overset{\text{def}}{=} \\ \{\texttt{pre}(s).x \to\ \texttt{let}\ (x, l) = x\ \texttt{in} \\ \quad PreBody(s, x, t, pat, \Gamma, \texttt{return}\ e\ \texttt{to}\ l) \end{array}$$

$$\begin{array}{l} PreBody(s, x, t, pat, \Gamma, e) \overset{\text{def}}{=} \\ \texttt{match}[t](\texttt{oneOf}(\texttt{pre}(s)); l; pat)\ \texttt{then}\ \texttt{args}(t, \Gamma, e)\ \texttt{else}\ x\} \end{array}$$

$$\begin{array}{l} Post(s, x, t, pat, \Gamma, e) \overset{\text{def}}{=} \\ \{\texttt{post}(s).x \to\ \texttt{let}\ (x, l) = x\ \texttt{in} \\ \texttt{match}[t](\texttt{oneOf}(\texttt{post}(s)); pat)\ \texttt{then}\ \texttt{args}(t, \Gamma, e)\ \texttt{else}\ x\} \end{array}$$

$$\begin{array}{l} \texttt{oneOf}(l_1, \ldots, l_n) \overset{\text{def}}{=} (l_1 \mid \ldots \mid l_n) \\ \texttt{args}(t, \Gamma, e) \overset{\text{def}}{=} \lambda a : t.\ \texttt{let}\ (\Gamma) = a\ \texttt{in}\ e \end{array}$$

Figure 10: MinAML Interpretation: "when" advice

The `within`$f(x)$ pattern requires that the top of the stack have the form $\texttt{val}:t :: f_{\texttt{post}}$; the variable x will be bound to the value matched by the pattern. The `cflow`(*pcd*) and `cflowtop`(*pcd*) clauses respectively compile to patterns that match the closest and farthest occurrence of *pat*.

The compilation of point-cut designators is similar to the translation shown in Figure 6. One difference is that the `when` clause pattern is matched before proceeding with the advice. The top of the stack for `before` advice must have the corresponding `pre` label; for `after` advice the top of the stack is the `post` label. Another difference is that the advice bodies bind the tuple of values extracted from the stack by the `match`. In the figure, the notation (Γ) stands for the tuple of variable bindings evident from the typing context Γ. Similarly, the notation $\Gamma \stackrel{\text{ctx}}{\Longrightarrow} t$ means that t is the tuple of types found in the context Γ.

This translation of MinAML is also type preserving. Let $\mathcal{P}(p:(t_1,t_2))$ be the context

$$p_{\texttt{pre}}:(t_1,t_2\ \texttt{label})\ \texttt{label}, p_{\texttt{post}}:t_2\ \texttt{label}$$

and let $\mathcal{P}(P)$ be the point-wise extension.

LEMMA 4.1 (TRANSLATION TYPE PRESERVATION).

1. *If* $P;\Gamma \vdash e : t \stackrel{\text{term}}{\Longrightarrow} e'$ *then* $\Gamma, \mathcal{P}(P) \vdash e' : t$.
2. *If* $P;\Gamma \vdash ds; e : t \stackrel{\text{decs}}{\Longrightarrow} e'$ *then* $\Gamma, \mathcal{P}(P) \vdash e' : t$.
3. *If* $P;\Gamma \vdash ad \stackrel{\text{adv}}{\Longrightarrow} e'$ *then* $\Gamma, \mathcal{P}(P) \vdash e' : \texttt{advice}$.
4. *If* $P;\Gamma \vdash pcd : \Gamma' \stackrel{\text{pcd}}{\Longrightarrow} pat$ *and* $\Gamma' \stackrel{\text{ctx}}{\Longrightarrow} t$ *then* $\Gamma, \mathcal{P}(P) \vdash pat : t$

5. Discussion

5.1 AspectML: A Prototype Implementation

In order to experiment further with our language design, we have developed a prototype implementation of most of the features described in this paper, omitting objects, and negation and value patterns for now. The prototype, which we call AspectML, is developed in SML/NJ [2] as an extension to core ML.

The core aspect calculus is implemented as a set of ML libraries for explicitly manipulating labeled program points, creating and using higher-order, first-class aspects and querying the calling context via a stack predicate language. The libraries have three main modules.

- `Point` manages creation and comparison of the structured labels used to mark join points.
- `RE` supplies utilities for building regular expression patterns out of points and matching them against point lists. It is implemented using the SML/NJ regular expression matching utilities.
- `Aspect` implements the operational semantics of the core calculus.

The library interfaces are included in Appendix C. For the most part, the implementation follows the theory directly. One deviation is that rather than limiting programmers to some set of domain-specific predicates for specifying point cuts, we have left the language open for experimentation. Programmers can use any function from label-stacks to Booleans as trigger predicates; if the function evaluates to true at a join point, the advice is invoked. Programmers can also expose the current label stack as a list of points and use regular expression queries to construct these functions or write their own functions over point lists.

One other deviation is that SML's type system is not quite strong enough to encode the heterogeneous list of aspects that makes up the aspect store.[4] Hence, before passing data to an aspect we need to coerce it into a universal data type (`UniversalDT.all`), and we need to coerce it back out of a universal data type on return.

The external language (AspectML) is implemented by a simple program rewriter that converts ".aml" files into ".sml" files. AspectML programmers may explicitly call the core calculus libraries if they wish to manipulate labeled program points directly. They may also use a new form of function declaration, `afun f (x:t1):t2 = e`, which implicitly allocates pre- and post-join points for monitoring function entry and exit as described earlier. Ordinary SML `fun` declarations cannot be monitored by aspects. Consequently, unlike in other aspect-oriented programming languages that we are aware of, AML programmers can choose to protect sections of their code from external interference and retain the standard ML reasoning principles that they are used to.

5.2 Related work

There are a number of aspect-oriented language design and implementation efforts that have already made a significant impact on industry, including AspectJ [10] and Hyper/J [13]. However, the study of the semantics of aspect-oriented languages lags well behind.

Most closely related to this paper is Tucker and Krishnamurthi's work on encoding aspects in Scheme [14]. Their approach uses *continuation marks*, a construct introduced by Clements et al. to aid in the implementation of program debugging tools [5]. Continuation marks are very similar to labeled program points except that (dynamically) they do not nest—the outer continuation mark overrides the inner. In the notation of this paper, the behavior of continuation marks could be modeled by adding an additional β rule: $l_1\langle l_2\langle v\rangle\rangle \longmapsto_\beta l_1\langle v\rangle$. This difference leads to a slightly more complex encoding of aspects. A more significant difference between this work and Tucker and Krishnamurthi's is that this paper develops a typed theory of aspects as opposed to an untyped theory of aspects.

Douence, Motelet and Sudholt [6] give a definition of point-cuts by encoding them in Haskell; they also provide an implementation in Java. However, the specification of advice is not integrated into their language. Instead, programs have two parts, an event (program point) producer and a monitor that consumes and reacts to these program points. Masuhara, Kiczales and Dutchyn [11] specify the semantics of an aspect-oriented language in Scheme and show how partial evaluation can be used to compile and optimize it.

A couple of authors have developed small, untyped formal calculi for reasoning about aspects. For instance, Wand, Kiczales and Dutchyn [16] have developed a denotational semantics for pointcuts and advice in a small aspect calcu-

[4] The aspect store would ideally be a list of $\exists t.t\ \texttt{label} \times t \to t$ elements. Unfortunately, SML does not provide existential types or the primitives for intensional type analysis that we would need. Stephanie Weirich suggested the encoding using universal data types that we currently use.

lus. Jagadeesen, Jeffrey and Riely [9] develop an object-oriented, aspect-oriented language and give a specification and correctness proof for weaving. In each case, join points are directly linked to the semantics of method calls rather than being developed as an orthogonal programming construct. We believe that elevating join points to the status of a first-class abstraction allows our semantic framework to be used in a broader collection of situations, including functional, imperative and object-oriented languages. We also feel that staging AOP semantics in terms of core and external languages is an important development as it helps modularize the theory and makes it possible to simplify the core language to its barest minimum.

Bauer, Ligatti and Walker [4] describe a language for constructing first-class and higher-order aspects. They also provide a system of logical combinators for composing advice and type and effect system to ensure that advice does not interfere with other advice. Unfortunately, the presence of aspect combinators makes the operational semantics for the language very complex. Consequently, their semantics does not make an appropriate platform for experimenting with aspect-oriented design in general or for investigating general-purpose reasoning principles in the presence of aspects.

Stack patterns provide a mechanism similar to stack inspection mechanism [15], and it would be interesting to explore this connection further. One might be able to implement stack-inspection-like security policies via aspects.

5.3 Future Work

Avenues for future research fall into four main categories:

1. Further development of the semantics of aspects.
2. Program analysis and type systems that promote safe use of aspect-oriented paradigms.
3. Extension of our AspectML implementation to cover all of Standard ML, including admitting further implicit program points and integration with ML's sophisticated module system.
4. Analysis of the performance cost of the advanced features including first-class advice and stack patterns.

While all of these directions are appealing, we plan to concentrate on items (1) and (3) in the near future. More specifically, we wish to consider enriching our simple calculus by adding new primitives, for example to explicitly delete advice. In addition, it seems desirable to develop a theory of contextual equivalence for aspect-oriented programs. We conjecture that such a theory will be tractable for the minimalist core calculus we presented in Section 2. With respect to item (3), we believe that defining a well-typed and well-scoped core aspect calculus is a significant step towards developing an aspect-oriented language that can interoperate benignly with advanced ML-style modules. In fact, our initial inspiration for labeled control-flow points was derived in part from the internal and external labels found in Harper and Lillibridge's translucent sum calculus [7].

6. Conclusions

This paper has shown that the main features of aspect-oriented languages can be modeled by a few relatively simple constructs in a core calculus. The key features are: labeled control flow points, support for manipulating data and control at those points, and a mechanism for inspecting the run-time stack. This approach leads to a (largely) language independent, semantically clean way of studying aspects. We have developed the theory of this core aspect calculus and demonstrated its applicability by type-directed translations from MinAML, a fragment of ML with aspects, and an object-oriented language. We claim that this approach is scalable, general, and theoretically well founded.

Acknowledgments

Many thanks to Dan Dantas, Kathleen Fisher, Stephanie Weirich, and the U. Penn. PL Club for their helpful feedback on earlier drafts of this work. We also thank Shriram Krishnamurthi and John Clements for pointing us to the work on aspects and continuation marks in Scheme.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, 1996.

[2] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In M. Wirsing, editor, *Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, New York, Aug. 1991. Springer-Verlag. Volume 528 of *Lecture Notes in Computer Science*.

[3] Aspect-oriented programming. In T. Elrad, R. E. Filman, and A. Bader, editors, *Special Issue of Communications of the ACM*, volume 40(10). Oct. 2001.

[4] L. Bauer, J. Ligatti, and D. Walker. Types and effects for non-interfereing program monitors. In *International Symposium on Software Security*, Tokyo, Japan, Nov. 2002.

[5] J. Clements, M. Flatt, and M. Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming*, pages 320–334, 2001.

[6] R. Douence, O. Motelet, and M. Südholt. A formal definition 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.

[7] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.

[8] 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.

[9] 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. To appear.

[10] 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.

[11] 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.

[12] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *ACM Conference on Functional Programming and Computer Architecture*, pages 66–77, La Jolla, June 1995.

[13] H. Ossher and P. Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *International conference on software engineering*, pages 734–737, Limerick, Ireland, June 2000.

[14] 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.

[15] D. S. Wallach, A. W. Appel, and E. W. Felten. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4), Oct. 2000.

[16] 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.

[17] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

APPENDIX

A. Core Language Summary

Types

$t ::= \texttt{bool} \mid t_1 \to t_2 \mid (t_1, \ldots, t_n)^{(n \geq 0)}$
$\quad \mid t\ \texttt{label} \mid t\ \texttt{pc} \mid \texttt{advice} \mid t\ \texttt{pat}$

Base calculus

$l \in$ Labels
$e ::= x \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3$
$\quad \mid \lambda x{:}t.\, e \mid e_1\ e_2$
$\quad \mid (e_1, \ldots, e_n)^{(n \geq 0)} \mid \texttt{let}\,(\vec{x}{:}\vec{t}) = e_1\, \texttt{in}\, e_2$

Aspects

$\quad \mid l \mid \{e_1.x \to e_2\} \mid e_1\langle e_2\rangle \mid \texttt{return}\ e_1\ \texttt{to}\ e_2$
$\quad \mid \texttt{new}\ x{:}t.\, e \mid e_1 \texttt{ >> } e_2 \mid e_1 \texttt{ << } e_2$

Label sets

$\quad \mid \{e_1, \ldots, e_n\} \mid e_1 \cup e_2 \mid e_1 \cap e_2$

Stack patterns

$\quad \mid e_1; e_2 \mid e_1 \mid e_2 \mid e^* \mid e_1 \,\&\, e_2 \mid \neg e \mid \texttt{val}{:}t$
$\quad \mid \texttt{match}[t](e)\ \texttt{then}\ e_1\ \texttt{else}\ e_2$
$\quad \mid \texttt{store}\ x{:}t = e_1\ \texttt{in}\ e_2$

B. MinAML Summary

Types

$t ::= \texttt{bool} \mid t_1 \to t_2 \mid [m_i{:}t_i]^{1..n}$

Expressions

$e ::= x \mid b \mid \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3$
$\quad \mid e.m \mid e_1.m \Leftarrow \varsigma\, x.e_2$
$\quad \mid \texttt{let}\, ds\, \texttt{in}\, e \mid e_1\, e_2$

Declarations

$ds ::= \cdot$
$\quad \mid (\texttt{bool}\, x = e)\, ds$
$\quad \mid (\texttt{fun}\, f(x{:}t_1){:}t_2 = e)\, ds$
$\quad \mid (\texttt{object}\, x{:}t = [m_i = \varsigma\, x_i.e_i]^{1..n})\ ds$
$\quad \mid \texttt{monitor}\, t.m\ ds$
$\quad \mid ad\ ds$

Program Points

$p ::= f \mid x.m \mid t.m$

Point Cuts

$pc ::= \{p_1, \ldots, p_n\}$

Point Cut Designators

$pcd ::= \texttt{within}f(x) \mid pcd_1\ \&\ pcd_2 \mid pcd_1 \mid pcd_2$
$\quad \mid \neg pcd \mid \texttt{cflow}(pcd) \mid \texttt{cflowtop}(pcd)$

Aspects

$ad ::= \texttt{before}\ pc(x)\ \texttt{when}\ pcd = e$
$\quad \mid \texttt{after}\ pc(x)\ \texttt{when}\ pcd = e$
$\quad \mid \texttt{around}\ pc(x)\ \texttt{when}\ pcd = e$
$\quad \mid \texttt{around}\ pc(x)\ \texttt{when}\ pcd = e_1; \texttt{proceed}\, y \to e_2$

C. AspectML Libraries

We have implemented a simple functional aspect-oriented programming language called AspectML (AML) as an extension of core SML/NJ. A simple rewriter translates AML files into well-typed SML/NJ source that can be compiled and linked to the libraries with the signatures below.

```
signature POINT = sig
  type point
  val new : string list -> point
  val toStrings : point -> string list
  val unique : point -> int
  val equals : point * point -> bool
 end

signature RE = sig
  type re
  (* functions to create base REs *)
  val primitive : Point.point -> re
  val empty : unit -> re
  (* functions to create complex REs *)
  val opt : re -> re   (* 0 or 1 of re *)
  val plus : re -> re   (* 1 or more of re *)
  val star : re -> re   (* 0 or more of re *)
  val concat : re -> re -> re  (* concatenation *)
  val alt : re -> re -> re  (* alternation *)
  (* attempts to match the RE to the point stack *)
  val match : re -> Point.point list -> bool
 end

signature ASPECT = sig
  type aspect
  type stack
  val toList : stack -> Point.point list
  (* create a new aspect *)
  val aspect : (stack -> bool)
      -> (stack * UniversalDT.all -> UniversalDT.all)
      -> aspect
  (* functions to add initial and final aspects *)
  val << : aspect -> unit
  val >> : aspect -> unit
  (* mark a control-flow point *)
  val mark : Point.point -> (unit -> UniversalDT.all)
      -> UniversalDT.all
  (* return a value from an aspect to a label *)
  val return : UniversalDT.all * Point.point -> 'a
 end
```