A Type-theoretic Interpretation of Pointcuts and Advice

Jay Ligatti
Princeton University
jligatti@cs.princeton.edu

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

Steve Zdancewic[†]
University of Pennsylvania
stevez@cis.upenn.edu

January 31, 2006

Abstract

This article defines the semantics of MinAML, an idealized aspect-oriented programming language, by giving a type-directed translation from a user-friendly external language to a 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 an elegant and extensible 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. We prove Preservation and Progress lemmas for our core language and show that the translation from MinAML source into core is type-preserving. Together these two results imply that the source language is type safe. We also consider several extensions to our basic framework including a general mechanism for analyzing the current call stack.

[*]This research was supported in part by National Science Foundation CAREER grant No. CCR-0238328, by ARDA Grant no. NBCHC030106, and by a Sloan Fellowship. Any opinions, findings and conclusions or recomendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, ARDA or Sloan foundation.

[†]This research is sponsored in part by the NSF Trusted Computing program, grant number CCR-0311204 "Dynamic Security Policies" and by the NSF CAREER award, grant number CNS03-46939 "Language-based Distributed System Security." Any opinions, findings and conclusions or recomendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

1 Introduction

Aspect-oriented programming languages (AOPL) [KLM$^+$97, Asp01], such as AspectJ [KHH$^+$01] and Hyper/J [OT00], 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 logging information, performance monitors, or security checks. An *aspect* is a collection of advice and corresponding join points that apply to a particular program.

The primary goal of this paper is to distill aspect-oriented programming into its fundamental components: (1) the join points, a means of designating "interesting" control-flow points, and (2) the advice, a way of manipulating the data and computation at those points. The objective is to obtain a simple, clear, and reusable semantic framework that researchers can use to explore new AOPL designs and to study the interactions between pointcuts, advice and more conventional language features. The main strength of our semantic framework is its simplicity. Since it is simple, it does not model all features found in full-scale AOPLs. For example, there is no obvious way to model Hyper/J's hyperslices [OT00].

One of the difficulties with specifying a simple and concise semantics for aspects is that according to Filman and Friedman's widely accepted definition [FF05], aspect-oriented programs must be *oblivious*. In other words, programmers should not be required to insert join point markers into their code manually. Manual insertion of the join point markers often leads to inconsistency, omission and other errors. Instead, the language *implicitly* associates join points with certain program constructs and the compiler is responsible for uniform insertion of these join points within programmer code. For example, object- and aspect-oriented languages normally specify that a join point exists immediately prior to execution of any method body and immediately after execution of any method body. Unfortunately, therefore, on the surface, the semantics of join points and advice is conflated with the semantics of objects and method invocation. Such a semantics breaks the principal of *orthogonality*, which suggests that each programming language construct should be understood independently of other programming language constructs. Tightly coupling join-point definitions with the semantics of methods and objects makes it impossible to understand aspects without first understanding methods and objects, which are complicated in isolation.

To resolve these difficulties, we adopt the central ideas of a type-theoretic semantic framework defined by Harper and Stone for Standard ML [HS98]. Rather than give a semantics directly to a large and relatively complex AOPL, we translate the unwieldy, but oblivious *external language* into a simple, unoblivious *core* *language* and then provide a precise and elegant operational semantics for the core. Though the core language is not oblivious, there is no reason to wish that it were — obliviousness is important in the source language used by programmers, but not at all necessary or desirable in our semantic intermediate language. The translation is beneficial as it compiles complex constructs into simpler ones and eliminates shallow syntactic differences between similar constructs. Overall, we believe it effectively modularizes the semantics.

Since we first presented our basic framework [WZL03], we have gained substantial additional experience with this style of semantics. In one case study, we explored the interaction between parametric polymorphism, intentional type analysis and aspects [DWWW05]. In a second case study, we equipped the core calculus with a type system for detecting and preventing interference between aspects and the mainline computation [DW04]. In both cases, our experience was very positive. We were able to define rich type systems and use our semantic framework to prove type safety results in the standard way. Most importantly, the complexity of our proofs in these cases was completely manageable. In the second case, we also proved a powerful non-interference result. The specifics of these extensions are well beyond the scope of this paper, but the experience is nonetheless valuable as it suggests that our semantic framework is robust enough for researchers to build upon in a variety of ways.

One possible disadvantage of our approach is that in order to establish certain correctness properties of source-language programs, it will be necessary to reason indirectly about the image of the translation of these programs in the core calculus. In some cases, this may be more difficult than establishing a direct semantics and reasoning about the source, though we have no definitive evidence either way. So far, for the type safety results we have proven and also for the non-interference result mentioned above, we have found the structure of the language definition has helped us modularize and simplify our proofs. Nevertheless, we do not expect our strategy to be the most effective in all cases.

In summary, the main contribution of this work is the definition of a novel type-theoretic framework for understanding aspects. Specific components of our theory include the following.

- A type-theoretic interpretation of an idealized 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. The main novelty of the core language are its two central abstractions:
    - Explicit, labeled join points that are defined orthogonally from the other constructs in the language, and
    - A single kind of first-class advice that, together with labeled join points, can give meaning to before, after and a simplified form of around advice.

- Several extensions to the basic framework including join point designators based on collections of labels and a mechanism for analysis of the current dynamic control context, which generalizes AspectJ's temporal operators. These extensions do not change the central machinery needed for aspects.

This work is an extended version of a paper that first appeared in the ACM SIGPLAN International Conference on Functional Programming [WZL03]. An important addition in this paper is the full proof of type safety for our core aspect calculus. These details demonstrate how simple and uncluttered the metatheory for our language is. We have also greatly improved the definition of context-sensitive advice by simplifying our mechanism for dynamic context analysis.

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 pointcut designators (Section 4). The paper concludes with a discussion of related work (Section 5) and some conclusions (Section 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 $v_1 + e\{v_2/x\}$. Triggering the advice associated with label l consumes the label; operationally this means that the advice runs only once per labeled point. 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. 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.[1]

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

$$\begin{array}{lcl} \{l.x \to x+1\} \texttt{ << } \{l.y \to y*2\} \texttt{ << } l\langle 3\rangle & \longmapsto^* & 7 \\ \{l.x \to x+1\} \texttt{ << } \{l.y \to y*2\} \texttt{ >> } l\langle 3\rangle & \longmapsto^* & 8 \end{array}$$

Evaluation proceeds from left to right. In the first example, the leftmost `<<` operator installs the advice $\{l.x \to x+1\}$; then, the rightmost `<<` installs the advice $\{l.y \to y*2\}$ *before* the other advice. Hence, the multiplication happens *before* the addition when the advice is triggered. The second example uses `>>` to install the aspects in the other order.

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. This permits labels as first class values that can be passed to and returned by functions. The example above can be rewritten as follows:

$$\texttt{new}\ p\!:\!\texttt{int}.\ \{p.x \to x+1\} \texttt{ << } \{p.y \to y*2\} \texttt{ << } p\langle 3\rangle$$

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

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

[1] One could imagine generalizations of this idea. For instance, one might want to augment the calculus with commands for uninstalling advice as well. This seems like a reasonable extension to the language, but we do not pursue it here as our simpler set of commands suffices for many interesting applications of aspects.

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 [WKD02] 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. This example illustrates the stateful nature of advice—the expression $\{ref.y \to y'\}$ `>>` `()` used in the *set* function evaluates to `()`, but installs the advice into the run-time environment as a side effect.

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

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 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.

$$\begin{array}{l}
\texttt{new}\ f_{\texttt{pre}}{:}\ \texttt{bool.} \\
\texttt{new}\ f_{\texttt{post}}{:}\ t\texttt{.} \\
\{f_{\texttt{pre}}.x \to \texttt{if}\ x\ \texttt{then}\ x\ \texttt{else}\ \texttt{return}\ e'\ \texttt{to}\ f_{\texttt{post}}\} \\
\texttt{>>} \\
\lambda x{:}\texttt{bool.}\,f_{\texttt{post}}\langle \texttt{let}\ x = f_{\texttt{pre}}\langle x\rangle\ \texttt{in}\ e\rangle
\end{array}$$

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 Syntax and 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, strings and n-tuples.

$$
\begin{array}{rcl}
l & \in & \text{Labels} \\
v & ::= & \{v.x \to e\} \mid b \mid g \mid \texttt{print} \mid l \mid \lambda x{:}t.\, e \mid \texttt{(}\vec{v}\texttt{)} \\
e & ::= & x \mid v \mid \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \mid e_1\; e_2 \\
 & \mid & \texttt{(}\vec{e}\texttt{)} \mid \texttt{let(}\vec{x}{:}\vec{t}\texttt{)} = e_1\, \texttt{in}\, e_2 \\
 & \mid & \texttt{new}\; x{:}t.\, e \mid e_1\langle e_2\rangle \mid \texttt{return}\, e_1\, \texttt{to}\, e_2 \\
 & \mid & \{e_1.x \to e_2\} \mid e_1 \;\texttt{>>}\; e_2 \mid e_1 \;\texttt{<<}\; e_2
\end{array}
$$

Let b range over the Boolean values `true` and `false`, g range over string values, 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). The operator `print` e prints its arguments e. If $\vec{e}$ is a vector of expressions $e_1, e_2, \ldots, e_n$ for $n \geq 0$, then $\texttt{(}\vec{e}\texttt{)}$ creates a tuple. The expression $\texttt{let(}\vec{x}:\vec{t}\texttt{)} = e_1\, \texttt{in}\, e_2$ binds the components of a tuple e_1 to the vector of variables $\vec{x}$ in the scope of e_2, and we often omit the parenthesis when the vector is of length one. Types on `let`-bound variables are often omitted when they are irrelevant or clear from context. To project the i^{th} component of a tuple, we often write $\pi_i e$, which is an abbreviation for $\texttt{let(}\vec{x}\texttt{)} = e\, \texttt{in}\, x_i$.

The pointcut language has been reduced to the barest minimum for the core calculus. However, the language design and semantics are completely compatible with more expressive pointcuts; Section 4 investigates several alternatives. Note that pointcuts, 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:

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

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 [MFH95] keeps track of a set L of labels (and their associated types). Similarly, A is an ordered list of installed advice—the `<<` and `>>` 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, $\langle L, A, e\rangle$.

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

Because the `return` operation needs to pass control to the nearest enclosing labeled point, it is convenient to define a function $\mathcal{S}(E)$ 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 $\mathcal{S}(E)$ is inductively defined on the structure of E, where the only interesting cases are:

$$\mathcal{S}([\ ]) = \cdot \qquad \mathcal{S}(l\langle E\rangle) = \mathcal{S}(E) :: l$$

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

$$\mathcal{S}(l_1\langle(\lambda x\!:\!t.\ l_3\langle e\rangle)\ l_2\langle[\ ]\rangle\rangle) = \cdot :: l_2 :: l_1$$

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.

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

The first five rules are the usual β-rules for a lambda calculus with Booleans, strings and tuples, where $e\{v/x\}$ is capture-avoiding substitution of the value v for the variable x in the expression e. We do not bother to model the output of the printing function; the reader will have to use their imagination. The sixth 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 pointcut 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 pointcuts. 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 pointcut satisfaction are defined by the following rules.

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

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

$$\frac{C \not\models v \quad \mathcal{A}[\![A]\!]_C = e'}{\mathcal{A}[\![\{v.x \to e\}, A]\!]_C = e'}$$

$$\frac{l = p}{\langle L, A, E[l\langle v\rangle]\rangle \models p}$$

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 $\mathcal{S}(-)$ function, this behavior is expressed by the following rule.

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

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. If there is no point labeled l in the context of the `return`, the `return` expression discards the entire context and the program terminates.

$$(l \notin \mathcal{S}(E)) \quad \langle L, A, E[\texttt{return}\, v\, \texttt{to}\, l]\rangle \longmapsto \langle L, A, \texttt{return}\, v\, \texttt{to}\, l\rangle$$

Figure 1 summarizes the operational rules for the core calculus.

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 pointcuts matching program contexts of type t, and `advice`, the type of advice. Types and typing contexts are given by the following grammar:

$$\begin{array}{rcl} t & ::= & \texttt{bool} \mid \texttt{string} \mid (\vec{t}) \mid t_1 \to t_2 \\ & \mid & t\,\texttt{label} \mid t\,\texttt{pc} \mid \texttt{advice} \\ \Gamma & ::= & \cdot \mid \Gamma, x{:}t \end{array}$$

The basic typing judgment has the form $\Gamma \vdash^{L} e : t$. It indicates that term e can be given type t in context Γ when labels have types given by L. Since the label typing L stays the same throughout a typing derivation, we normally omit it from the judgment and simply write $\Gamma \vdash e : t$.

Figure 2 contains the typing rules for the aspect calculus. The first three lines in the figure give typing rules for booleans, functions and tuple typing. They are completely standard. The last four lines in the figure give typing rules for labels, pointcuts and advice. 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 pointcuts 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 pointcut 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 pointcut 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.

$\boxed{\text{C} \longmapsto_\beta \text{C'}}$

$$\begin{array}{rl}
\langle L, A, (\lambda x\!:\!t.\ e)\ v\rangle & \longmapsto_\beta \langle L, A, e\{v/x\}\rangle \\
\langle L, A, \texttt{if}\,\texttt{true}\,\texttt{then}\,e_1\,\texttt{else}\,e_2\rangle & \longmapsto_\beta \langle L, A, e_1\rangle \\
\langle L, A, \texttt{if}\,\texttt{false}\,\texttt{then}\,e_1\,\texttt{else}\,e_2\rangle & \longmapsto_\beta \langle L, A, e_2\rangle \\
\langle L, A, \texttt{print}\ g\rangle & \longmapsto_\beta \langle L, A, \texttt{()}\rangle \\
\langle L, A, \texttt{let}\ \texttt{(}\vec{x}\!:\!\vec{t}\texttt{)} = \texttt{(}\vec{v}\texttt{)}\ \texttt{in}\ e\rangle & \longmapsto_\beta \langle L, A, e\{\vec{v}/\vec{x}\}\rangle \\
(l \notin L) \quad \langle L, A, \texttt{new}\ x\!:\!t.\ e\rangle & \longmapsto_\beta \langle (L, l\!:\!t), A, e\{l/x\}\rangle \\
\langle L, A, a\ \texttt{<<}\ e\rangle & \longmapsto_\beta \langle L, (a, A), e\rangle \\
\langle L, A, a\ \texttt{>>}\ e\rangle & \longmapsto_\beta \langle L, (A, a), e\rangle \\
(l \notin \mathcal{S}(E)) \quad \langle L, A, l\langle E[\texttt{return}\ v\ \texttt{to}\ l]\rangle\rangle & \longmapsto_\beta \langle L, A, l\langle v\rangle\rangle
\end{array}$$

$\boxed{\text{C} \longmapsto \text{C'}}$

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

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

$$(l \notin \mathcal{S}(E)) \quad \langle L, A, E[\texttt{return}\ v\ \texttt{to}\ l]\rangle \longmapsto \langle L, A, \texttt{return}\ v\ \texttt{to}\ l\rangle$$

$\boxed{\mathcal{A}[\![A]\!]_C = e}$

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

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

$$\frac{C \not\models v \quad \mathcal{A}[\![A]\!]_C = e'}{\mathcal{A}[\![\{v.x \to e\}, A]\!]_C = e'}$$

$\boxed{\text{C} \models \text{p}}$

$$\frac{l = p}{\langle L, A, E[l\langle v\rangle]\rangle \models p}$$

Figure 1: Core Calculus Operational Semantics

$\boxed{\Gamma \vdash e : t}$

$$\frac{}{\Gamma \vdash b : \texttt{bool}} \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : t' \quad \Gamma \vdash e_3 : t'}{\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : t'}$$

$$\frac{}{\Gamma \vdash g : \texttt{string}} \qquad \frac{\Gamma \vdash e : \texttt{string}}{\Gamma \vdash \texttt{print}\, e : ()}$$

$$\frac{\Gamma \vdash e_i : t_i \quad (1 \leq i \leq n)}{\Gamma \vdash (e_1, \ldots, e_n)^{(n \geq 0)} : (t_1, \ldots, t_n)^{(n \geq 0)}} \qquad \frac{\Gamma \vdash e_1 : (\vec{t}) \quad \Gamma, \vec{x}{:}\vec{t} \vdash e_2 : t'}{\Gamma \vdash \texttt{let}\ (\vec{x}{:}\vec{t}) = e_1\ \texttt{in}\ e_2 : t'}$$

$$\frac{\Gamma, x{:}t \vdash e : t'}{\Gamma \vdash \lambda x.e : t \to t'} \qquad \frac{\Gamma \vdash e_1 : t \to t' \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1\ e_2 : t'}$$

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

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

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

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

Figure 2: Core Calculus Type system

Installing advice only causes side effects to the run-time environment, so all advice expressions, regardless of the type of data they manipulate, can be given the type `advice`. The rules for installing advice using `<<` and `>>` 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.

2.3 Type Safety

The typing rules are sound with respect to the operational semantics, and our language design leads to a soundness proof in the style of Wright and Felleisen [WF94]. Our proof is quite straightforward, so readers familiar with type safety proofs may want to skip this section and move quickly onto the next.

The first step is to prove the standard substitution lemma by induction on the structure of the typing derivation for expressions.

Lemma 1 (Substitution)
If $\Gamma, x{:}t' \vdash e : t$ *and* $\Gamma \vdash e' : t'$ *then* $\Gamma \vdash e[e'/x] : t$.

The following two lemmas are also essential for proving type safety. The first, Weakening, may be proven by induction on the structure of the typing derivation.[2] The second, Inversion of Typing, or simply Inversion, follows directly by inspection of the typing rules.

Lemma 2 (Weakening)
If $\Gamma \vdash^{L} e : t'$ *and* L' *extends* L *and* Γ' *extends* Γ *then* $\Gamma' \vdash^{L'} e : t'$.

Lemma 3 (Inversion of Typing)
Every typing rule is invertible. In other words, if the conclusion of a particular rule holds, then its premises must also hold. For example, if $\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : t'$ *then* $\Gamma \vdash e_1 : \texttt{bool}$ *and* $\Gamma \vdash e_2 : t'$ *and* $\Gamma \vdash e_3 : t'$.

Next, we determine some of the properties of the values that inhabit each type by proving a canonical forms lemma. This lemma follows by induction on the structure of the typing derivations for values.

Lemma 4 (Canonical Forms)
If $\cdot \vdash^{L} v : t$ *then*

- $t = \texttt{bool}$ *implies* v *is a boolean* b,
- $t = \texttt{string}$ *implies* v *is a string* g,
- $t = (t_1, \ldots, t_n)$ *implies* v *is* $(v_1, \ldots, v_n)$,
- $t = t_1 \rightarrow t_2$ *implies* v *is* $\lambda x{:}t_1.e$,
- $t = t'\ \texttt{label}$ *implies* v *is* l
- $t = t'\ \texttt{pc}$ *implies* v *is* l, *and*
- $t = \texttt{advice}$ *implies* v *is* $\{e_1.x \rightarrow e_2\}$

Well-typed computational contexts also have important properties that we use in the safety proof. The Well-typed Filled Context Lemma is a corollary of the definition of well-typed contexts and the Substitution Lemma. Both the Decomposition lemmas can be proven by induction on the typing derivation for expressions.

Definition 5 (Well-typed Context)
A context E *is well typed, written* $\Gamma \vdash E : t \Rightarrow t'$, *if* $x \notin FV(E)$ *and* $\Gamma, x{:}t \vdash E[x] : t'$.

[2]We do not distinguish between contexts L and Γ that only differ in the order of assumptions, so judgments containing them automatically satisfy the exchange property. They also satisfy other standard structural properties that are not necessary for our purposes here.

Corollary 6 (Well-typed Filled Context)
If $\Gamma \vdash E : t \Rightarrow t'$ and $\Gamma \vdash e : t$ then $\Gamma \vdash E[e] : t'$

Lemma 7 (Decomposition I)
If $\cdot \vdash e : t$ then either

1. *e is a value v, or*
2. *e can be decomposed into $E[r]$ where r is a redex that can be reduced immediately by one of the $\longmapsto_\beta$ reductions or r has the form `return` v `to` l.*

Lemma 8 (Decomposition II)
If $\cdot \vdash E[e] : t'$ then there exists a type t such that $\cdot \vdash E : t \Rightarrow t'$ and $\cdot \vdash e : t$.

Now, before we move on to the final results, we must specify what it means for our abstract machine to be well-typed and for execution to be successfully completed (i.e., *finished*)

Definition 9 (Finished Configuration)
A finished configuration is of the form $\langle L, A, v\rangle$ or the form $\langle L, A,$ `return` v `to` $l\rangle$.

The first kind of finished configuration reprepresents normal termination of a program that has computed the value v; the second kind of finished configuration represents abnormal termination in which a value is returned to a labeled point that is not in scope.

Definition 10 (Well-typed Configuration)
A configuration $\langle L, A, e\rangle$ is well typed, written $\vdash \langle L, A, e\rangle$ `ok`, if, for all advice $a \in A$, it is the case that $\cdot \vdash^L a :$ `advice`, and $\cdot \vdash^L e : t$ for some t.

Finally, we have enough information to state and prove the progress lemma.

Theorem 11 (Progress)
If $\vdash C$ `ok` then either the configuration is finished, or there exists another configuration C' such that $C \longmapsto C'$.

Proof Let $C = \langle L, A, e\rangle$. Since e is well-typed, by Decomposition I, it is either (1) a value v, or it has the form $E[r]$ where (2) r is a redex that can be reduced immediately by one of the $\longmapsto_\beta$ reductions, or (3) r has the form `return` v `to` l. In case (1), the configuration is finished. In case (2), the configuration can take a step using the context rule.

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

In case (3), the context E may be empty, in which case the configuration is finished. Otherwise, a reduction can take place via one of the two rules for the `return` statement. More specifically, either $E = E'[l\langle E''[\;]\rangle]$ and $l \notin \mathcal{S}(E'')$, in which case

$$\frac{(l \notin \mathcal{S}(E'')) \quad \langle L, A, l\langle E''[\texttt{return}\ v\ \texttt{to}\ l]\rangle\rangle \longmapsto_\beta \langle L, A, l\langle v\rangle\rangle}{\langle L, A, E'[l\langle E''[\texttt{return}\ v\ \texttt{to}\ l]\rangle]\rangle \longmapsto \langle L', A', E'[l\langle v\rangle]\rangle}$$

or, $l \notin \mathcal{S}(E)$ and,

$$(l \notin \mathcal{S}(E)) \quad \langle L, A, E[\texttt{return}\ v\ \texttt{to}\ l]\rangle \longmapsto \langle L, A, \texttt{return}\ v\ \texttt{to}\ l\rangle$$

■

Our last step is to prove the preservation lemma for the language. To do that, we need two additional minor lemmas concerning the composition of advice.

Lemma 12 (Well-typed Advice Selection)
Let $C = \langle L, A, E[l\langle v'\rangle]\rangle$. If $C \models v$ and $\cdot \vdash^L \{v.x \to e\} : \texttt{advice}$ and $L(l) = t$ then $x\!:\!t \vdash^L e : t$.

Proof This fact comes directly from the definition of the matching judgment and inversion of the typing rules. ■

Lemma 13 (Well-typed Advice Composition)
If $\mathcal{A}[\![\cdot]\!]_{\langle L,A,E[l\langle v'\rangle]\rangle} = e'$ then $\cdot \vdash^L e' : L(l) \to L(l)$.

Proof By induction on the definition of the advice composition judgment, using the Well-typed Advice Selection Lemma. ■

Theorem 14 (β-Preservation)
If $\vdash \langle L, A, e\rangle$ `ok` and $\langle L, A, e\rangle \longmapsto_\beta \langle L', A', e'\rangle$ then L' extends L and there is a derivation of $\vdash \langle L', A', e'\rangle$ `ok`.

Proof The proof is by cases on the operational rules. All of the cases are straightforward. There is one slight subtlety in the case for new labels: We must use the Weakening lemma to show that the stored advice in A remains well-typed when we add a new label L to the store.

Here is the case for the operation of the `return` statement:

- Given: $\langle L, A, l\langle E[\texttt{return}\ v\ \texttt{to}\ l]\rangle\rangle \longmapsto_\beta \langle L, A, l\langle v\rangle\rangle$ when $l \notin \mathcal{S}(E)$.
  Since $\vdash \langle L, A, l\langle E[\texttt{return}\ v\ \texttt{to}\ l]\rangle\rangle$ `ok`, we have

  (1) for all $a \in A$, $\cdot \vdash^L a : \texttt{advice}$, and

  (2) $\cdot \vdash^L l\langle E[\texttt{return}\ v\ \texttt{to}\ l]\rangle : t$ for some t.

  From (2), and by inversion of the typing rules, we can conclude that

  (3) $L(l) = t$, and

(4) $\cdot \vdash^L E[\texttt{return}\, v\, \texttt{to}\, l] : t$.

From (4), and by Decomposition II, we conclude that $\cdot \vdash^L \texttt{return}\, v\, \texttt{to}\, l : t'$, for some t'. By inversion of typing and (3), we know that $\cdot \vdash^L v : t$. Consequently, due to the typing rule for labels, we can conclude

(5) $\cdot \vdash^L l\langle v\rangle : t$.

From (1) and (5), we have our result: $\vdash \langle L, A, l\langle v\rangle\rangle$ `ok` (and L trivially extends itself).

■

Theorem 15 (Preservation)
If $\vdash \langle L, A, e\rangle$ `ok` *and* $\langle L, A, e\rangle \longmapsto \langle L', A', e'\rangle$ *then* L' *extends* L *and there is a derivation of* $\vdash \langle L', A', e'\rangle$ `ok`.

Proof The proof is by cases on the operational rules. All three cases are straightforward.

- The first case concerns β-reduction inside a computational context. It follows directly from β-preservation and the Well-typed Filled Context Corollary.
- The second case concerns application of advice. It follows directly from the Well-typed Advice Composition Lemma and the Well-typed Filled Context Corollary.
- The third case concerns the `return` statement. The proof here is analogous to the proof involving `return` in the β-preservation lemma given above.

■

3 MinAML

This section gives a semantics for a concrete AOPL called MinAML by translating it into the core aspect calculus. Figure 3 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.

MinAML has three sorts of aspects: those that give advice *before* execution of pointcut p (for now, p is limited to be a function call), those that give advice

$$
\begin{array}{llcl}
\text{types} & t & ::= & \texttt{bool} \mid \texttt{string} \mid \texttt{unit} \mid t_1 \rightarrow t_2 \\
\text{terms} & e & ::= & x \mid b \mid g \mid \texttt{()} \mid \texttt{if}\, e_1 \,\texttt{then}\, e_2 \,\texttt{else}\, e_3 \\
 & & \mid & \texttt{print}\; e \mid \texttt{let}\, ds \,\texttt{in}\, e \mid e_1\, e_2 \\
\text{decls} & ds & ::= & \cdot \\
 & & \mid & \texttt{(bool}\, x = e\texttt{)}\; ds \\
 & & \mid & \texttt{(string}\, x = e\texttt{)}\; ds \\
 & & \mid & \texttt{(unit}\, x = e\texttt{)}\; ds \\
 & & \mid & \texttt{(fun}\, f\,\texttt{(}x\texttt{:}t_1\texttt{):}t_2 = e\texttt{)}\; ds \\
 & & \mid & ad\; ds \\
\text{prog pts} & p & ::= & f \\
\text{aspects} & ad & ::= & \texttt{before}\; p(x, fn) = e \\
 & & \mid & \texttt{after}\; p(x, fn) = e \\
 & & \mid & \texttt{around}\; p(x, fn) = e \\
 & & \mid & \texttt{around}\; p(x, fn) = e_1; \texttt{proceed}\, y \rightarrow e_2
\end{array}
$$

Figure 3: MinAML Syntax

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. In the latter case, after proceeding with p, a fresh variable y is bound to the result of the function. All forms of advice have access to metadata associated with the join point that triggered it. This metadata is passed to the advice via the auxiliary parameter fn. For our current purposes, we assume the metadata is a string containing the name of the function from the source text. However, there are other useful forms of metadata, such as access control privileges, that one might wish to assign to a join point. Our semantics is flexible enough to accommodate any sort of metadata one might be interested in.

Note that the around advice we present here is not as general as the around advice found in AspectJ. In AspectJ, the *proceed* statement may appear anywhere within the body of the around advice and may even appear multiple times. Moreover, as shown recently by Clifton and Leavens [CL05], a full semantics for AspectJ-style around advice with *proceed* is substantially more complicated. Consequently, rather than attempt to model full AspectJ-style advice, we satisfy ourselves with this very simple substitute.

MinAML also deviates from AspectJ in another important way. AspectJ allows programmers to refer to any method that appears anywhere in their program, even private methods of classes. In contrast, 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 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 judgments for terms, for declarations and for advice. The term translation judgment has the form $P; \Gamma \vdash e : t \overset{\mathsf{term}}{\Longrightarrow} e'$. 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 $f : \texttt{bool} \rightarrow \texttt{int}$ extends the context P with the binding $f : (\texttt{bool}, \texttt{int})$ and extends the typing context Γ with $f : \texttt{bool} \rightarrow \texttt{int}$.

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 4, 5 and 6 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 ($f_{\texttt{pre}}$) and one for the exit point ($f_{\texttt{post}}$). 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 $p_{\texttt{pre}} : (t_1, \texttt{string})\ \texttt{label}, p_{\texttt{post}} : (t_2, \texttt{string})\ \texttt{label}$. The type `string` appears here since advice receives metadata from each join point. As discussed in the previous subsection, this metadata is the source-text string name of the function. We assume the presence of an unspecified function $\mathcal{M}(f)$ to generate this metadata for us during the translation.

The key 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 $p_{\texttt{pre}}$ join point. *After* advice for p is triggered by the $p_{\texttt{post}}$ join point. *Around* advice with a proceed statement defines two pieces of advice, one

[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 [HL94]. Once naming conventions for the external language have been overcome, the translation to internal language should be straightforward.

$$\boxed{P; \Gamma \vdash e : t \overset{\textsf{term}}{\Longrightarrow} e'}$$

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

$$\frac{}{P; \Gamma \vdash \texttt{()} : \texttt{unit} \overset{\textsf{term}}{\Longrightarrow} \texttt{()}} \qquad \frac{P; \Gamma \vdash e : \texttt{string} \overset{\textsf{term}}{\Longrightarrow} e'}{P; \Gamma \vdash \texttt{print}\, e : \texttt{unit} \overset{\textsf{term}}{\Longrightarrow} \texttt{print}\, e'}$$

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

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

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

Figure 4: MinAML Interpretation: Terms

$\boxed{P;\Gamma \vdash ds; e : t \xRightarrow{\textsf{decs}} e'}$

$$\frac{P;\Gamma \vdash e : t \xRightarrow{\textsf{term}} e'}{P;\Gamma \vdash \cdot; e : t \xRightarrow{\textsf{decs}} e'}$$

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

$$\frac{P;\Gamma \vdash e_1 : \texttt{string} \xRightarrow{\textsf{term}} e_1' \quad P;\Gamma, x{:}\texttt{string} \vdash ds; e_2 : t \xRightarrow{\textsf{decs}} e_2'}{P;\Gamma \vdash \texttt{(string}\, x = e_1\texttt{)}\; ds; e_2 : t \xRightarrow{\textsf{decs}} \texttt{let}\, x{:}\texttt{string} = e_1'\, \texttt{in}\, e_2'}$$

$$\frac{P;\Gamma \vdash e_1 : \texttt{unit} \xRightarrow{\textsf{term}} e_1' \quad P;\Gamma, x{:}\texttt{unit} \vdash ds; e_2 : t \xRightarrow{\textsf{decs}} e_2'}{P;\Gamma \vdash \texttt{(unit}\, x = e_1\texttt{)}\; ds; e_2 : t \xRightarrow{\textsf{decs}} \texttt{let}\, x{:}\texttt{unit} = e_1'\, \texttt{in}\, e_2'}$$

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

$$\text{where } e_b = \lambda x{:}t_1.\pi_1\; (f_{\texttt{post}}\langle\; \begin{array}{l} \texttt{let}\, a = (x, \mathcal{M}(f))\,\texttt{in} \\ \texttt{let}\, x = \pi_1\; f_{\texttt{pre}}\langle a\rangle\; \texttt{in}\,(e_1', \pi_2\; a)\; \rangle) \end{array}$$

$$\frac{P;\Gamma \vdash ad \xRightarrow{\textsf{adv}} e_1' \quad P;\Gamma \vdash ds; e_2 : t \xRightarrow{\textsf{decs}} e_2'}{P;\Gamma \vdash ad\; ds; e_2 : t \xRightarrow{\textsf{decs}} e_1' \;\texttt{>>}\; e_2'}$$

Figure 5: MinAML Interpretation: Declarations

$$\boxed{P;\Gamma \vdash ad \overset{\mathsf{adv}}{\Longrightarrow} e'}$$

$$\frac{p\!:\!(t_1,t_2) \in P \quad P;\Gamma, x\!:\!t_1, \mathit{fn}\!:\!\mathtt{string} \vdash e : t_1 \overset{\mathsf{term}}{\Longrightarrow} e'}{P;\Gamma \vdash \mathtt{before}\ p(x,\mathit{fn}) = e \overset{\mathsf{adv}}{\Longrightarrow} \{p_{\mathtt{pre}}.x \to \mathtt{let}\ (x,\mathit{fn}) = x\ \mathtt{in}\ (e',\mathit{fn})\}}$$

$$\frac{p\!:\!(t_1,t_2) \in P \quad P;\Gamma, x\!:\!t_2, \mathit{fn}\!:\!\mathtt{string} \vdash e : t_2 \overset{\mathsf{term}}{\Longrightarrow} e'}{P;\Gamma \vdash \mathtt{after}\ p(x,\mathit{fn}) = e \overset{\mathsf{adv}}{\Longrightarrow} \{p_{\mathtt{post}}.x \to \mathtt{let}\ (x,\mathit{fn}) = x\ \mathtt{in}\ (e',\mathit{fn})\}}$$

$$\frac{\begin{array}{c} p\!:\!(t_1,t_2) \in P \\ P;\Gamma, x\!:\!t_1, \mathit{fn}\!:\!\mathtt{string} \vdash e_1 : t_1 \overset{\mathsf{term}}{\Longrightarrow} e_1' \\ P;\Gamma, y\!:\!t_2, \mathit{fn}\!:\!\mathtt{string} \vdash e_2 : t_2 \overset{\mathsf{term}}{\Longrightarrow} e_2' \end{array}}{\begin{array}{l} P;\Gamma \vdash \mathtt{around}\ p(x,\mathit{fn}) = e_1; \mathtt{proceed}\, y \to e_2 \overset{\mathsf{adv}}{\Longrightarrow} \\ \quad \{p_{\mathtt{pre}}.x \to \mathtt{let}\ (x,\mathit{fn}) = x\ \mathtt{in}\ (e_1',\mathit{fn})\}\ \mathtt{>>} \\ \quad\ \{p_{\mathtt{post}}.y \to \mathtt{let}\ (y,\mathit{fn}) = y\ \mathtt{in}\ (e_2',\mathit{fn})\} \end{array}}$$

$$\frac{p\!:\!(t_1,t_2) \in P \quad P;\Gamma, x\!:\!t_1, \mathit{fn}\!:\!\mathtt{string} \vdash e : t_2 \overset{\mathsf{term}}{\Longrightarrow} e'}{\begin{array}{c} P;\Gamma \vdash \mathtt{around}\ p(x,\mathit{fn}) = e \overset{\mathsf{adv}}{\Longrightarrow} \\ \{p_{\mathtt{pre}}.x \to \mathtt{let}\ (x,\mathit{fn}) = x\ \mathtt{in}\ \mathtt{return}\ (e',\mathit{fn})\ \mathtt{to}\ p_{\mathtt{post}}\} \end{array}}$$

Figure 6: MinAML Interpretation: Aspects

for the $p_{\mathtt{pre}}$ point and one for the $p_{\mathtt{post}}$ point. Finally, *around* advice without a proceed statement is triggered by $p_{\mathtt{pre}}$ but returns to $p_{\mathtt{post}}$.

Importantly, this translation produces well-typed core language terms: Let $\mathcal{P}(p:(t_1,t_2))$ be the context $p_{\mathtt{pre}} : (t_1, \mathtt{string})\ \mathtt{label}, p_{\mathtt{post}} : (t_2, \mathtt{string})\ \mathtt{label}$ and let $\mathcal{P}(P)$ be the point-wise extension of the former translation.

Lemma 16 (Translation Type Preservation)

1. *If* $P;\Gamma \vdash e : t \overset{\mathsf{term}}{\Longrightarrow} e'$ *then* $\Gamma, \mathcal{P}(P) \vdash e' : t$.

2. *If* $P;\Gamma \vdash ds; e : t \overset{\mathsf{decs}}{\Longrightarrow} e'$ *then* $\Gamma, \mathcal{P}(P) \vdash e' : t$.

3. *If* $P;\Gamma \vdash ad \overset{\mathsf{adv}}{\Longrightarrow} e'$ *then* $\Gamma, \mathcal{P}(P) \vdash e' : \mathtt{advice}$.

The proof of Lemma 16 is by induction on the translation derivation. Combining Lemma 16 with the type safety result for the core language yields an important safety result for MinAML.

Theorem 17 (MinAML Safety)
Suppose that $\cdot;\cdot \vdash e : t \overset{\mathsf{term}}{\Longrightarrow} e'$. *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 [AC96]. 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 some evidence that the semantic framework can be used in a variety of different computational contexts.

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 [AC96].

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

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.

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

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

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

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 $x.m$.

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

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,fn) = 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.

4 Complex Pointcuts

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 advice code to be shared by many program points. The second generalization is to allow run-time inspection of the labels that appear in the call stack, which permits 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 $t_1 \rightarrow t_2$ 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:

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

The advice $\{\{l_1, \ldots, l_n\}.x \rightarrow e\}$ 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 pointcuts. 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 imagine further refinements along these lines. In particular, since all labels in the set must have the same type, it is impossible to construct pointcuts that represent all labels or all labels from a particular module. Such a facility is useful when one wants to write a single piece of advice that instruments many control flow points with tracing or profiling information. While it is beyond the scope of this paper, it is possible to develop this calculus with the ability to construct polymorphic pointcuts and polymorphic advice [DWWW05].

4.1.1 MinAML Extensions and Interpretation

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

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

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 to which it should return.

The new translation of function and advice declarations appears in Figure 7. 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.

4.2 Context analysis

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 logging is to print the arguments of a function f when it is called from inside the body of a second function, g. The

$$\boxed{P;\Gamma \vdash ds;e : t \overset{\mathsf{decs}}{\Longrightarrow} e'}$$

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

$$\text{where } e_b = \lambda x\!:\!t_1.\pi_1\; (f_{\texttt{post}}\langle \begin{array}{l} \texttt{let}\, a = (x, \mathcal{M}(f), f_{\texttt{post}})\, \texttt{in} \\ \texttt{let}\, x = \pi_1\; f_{\texttt{pre}}\langle a\rangle\; \texttt{in}(e_1', \pi_2\; a)\; \rangle) \end{array}$$

$$\boxed{P;\Gamma \vdash ad \overset{\mathsf{adv}}{\Longrightarrow} e'}$$

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

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

$$\frac{\begin{array}{c} (p\!:\!(t_1,t_2) \in P)^{p\in s} \\ P;\Gamma, x\!:\!t_1, \mathit{fn}\!:\!\texttt{string} \vdash e_1 : t_1 \overset{\mathsf{term}}{\Longrightarrow} e_1' \\ P;\Gamma, y\!:\!t_2, \mathit{fn}\!:\!\texttt{string} \vdash e_2 : t_2 \overset{\mathsf{term}}{\Longrightarrow} e_2' \end{array}}{\begin{array}{l} P;\Gamma \vdash \texttt{around}\; s(x,\mathit{fn}) = e_1; \texttt{proceed}\, y \to e_2 \overset{\mathsf{adv}}{\Longrightarrow} \\ \quad \{\texttt{pre}(s).x \to \texttt{let}\;(x,\mathit{fn},l) = x\; \texttt{in}\;(e_1',\mathit{fn},l)\} \texttt{>>} \\ \qquad \{\texttt{post}(s).y \to \texttt{let}\;(y,\mathit{fn}) = y\; \texttt{in}\;(e_2',\mathit{fn})\} \end{array}}$$

$$\frac{(p\!:\!(t_1,t_2) \in P)^{p\in s} \quad P;\Gamma, x\!:\!t_1, \mathit{fn}\!:\!\texttt{string} \vdash e : t_2 \overset{\mathsf{term}}{\Longrightarrow} e'}{\begin{array}{c} P;\Gamma \vdash \texttt{around}\; s(x,\mathit{fn}) = e \overset{\mathsf{adv}}{\Longrightarrow} \\ \{\texttt{pre}(s).x \to \texttt{let}\;(x,\mathit{fn},l) = x\; \texttt{in}\; \texttt{return}\,(e',\mathit{fn})\; \texttt{to}\; l\} \end{array}}$$

Figure 7: MinAML Interpretation: Label Sets

logging 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 to the advice construct itself, but it is simpler to provide an orthogonal mechanism for querying the run-time state of the program. This section presents stack analysis as a mechanism that achieves the desired expressiveness without altering the basic functionality of advice. We developed this mechanism with the help of Dan Dantas, who uses something similar in his system of harmless advice [DW04]. It is also possible to develop a polymorphic version, as Dantas, Walker, Washburn and Weirich [DWWW05] have demonstrated.

During the course of evaluation, 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 where to jump to. Stack analysis allows programmers to write queries over the label stack.

Aspect-oriented languages also permit queries on the *data* stored in the run-time stack. To handle this feature, we extend the core language with a means of storing values on the stack by adding a new expression $\texttt{store}\ x{:}t = e_1\ \texttt{in}\ e_2$. The semantics of the `store` expression resembles the semantics of `let`, in that e_1 is evaluated first and bound to x before e_2 is evaluated. The difference is that we retain value v_1 that results from evaluation of e_1 "on the stack" as we execute e_2. We formalize this behavior by introducing a second syntactic form $\texttt{stored}\ v_1{:}t\ \texttt{in}\ e$ that remembers v_1 as we execute e.

$$\begin{array}{l} e ::= \ldots \mid \texttt{store}\ x{:}t = e_1\ \texttt{in}\ e_2 \mid \texttt{stored}\ v{:}t\ \texttt{in}\ e \\ E ::= \ldots \mid \texttt{store}\ x{:}t = E\ \texttt{in}\ e \mid \texttt{stored}\ v{:}t\ \texttt{in}\ E \end{array}$$

Notice that the context $\texttt{stored}\ v{:}t\ \texttt{in}\ E$ allows evaluation under the binding, though before introducing the stored command, x is substituted away. The following two β-rules make these ideas precise.

$$\begin{array}{c} \langle L, A, \texttt{store}\ x{:}t = v\ \texttt{in}\ e\rangle \longmapsto_\beta \langle L, A, \texttt{stored}\ v{:}t\ \texttt{in}\ e[v/x]\rangle \\ \langle L, A, \texttt{stored}\ v{:}t\ \texttt{in}\ v'\rangle \longmapsto_\beta \langle L, A, v'\rangle \end{array}$$

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

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

The current run-time stack is reified as a first-class data structure via the `stack()` expression. Another expression, $\texttt{stkcase}\ e_1\ (pat \to e_2\ \texttt{|}\ \_ \to e_3)$, allows programs to pattern match against a stack: after e_1 evaluates to stack s, evaluation proceeds either with e_2 (if s matches pat) or e_3 (otherwise).

$$\begin{array}{rcl}
v & ::= \dots & \mid\ s \\
\tau & ::= \dots & \mid\ \texttt{stack} \\
e & ::= \dots & \mid\ \texttt{stack()}\ \mid\ \texttt{stkcase}\, e_1\, (pat \to e_2\ \texttt{|}\ \_ \to e_3) \\
E & ::= \dots & \mid\ \texttt{stkcase}\, E\, (pat \to e_2\ \texttt{|}\ \_ \to e_3)
\end{array}$$

Stack patterns describe the stack of labels and stored values present in the dynamic evaluation context. There are seven forms of stack patterns, inductively defined according to the following grammar.

$$\begin{array}{rl}
pat\ ::= & \cdot\ \mid\ x\!:\!\texttt{stack}\ \mid\ \texttt{wild} :: pat\ \mid\ \texttt{val}\!:\!t = v :: pat\ \mid \\
& \texttt{val}\!:\!t = x :: pat\ \mid\ l :: pat\ \mid\ x\!:\!t\ \texttt{label} :: pat
\end{array}$$

Most of these patterns are self-explanatory. The $\cdot$ pattern matches exactly the empty stack $\cdot$, while $x : \texttt{stack}$ matches every stack and binds the matched stack to the variable x in the `stkcase` expression. Using `wild` in a pattern allows any single element on the stack to be matched. Data and label values can be explicitly provided in stack patterns, such as in the pattern $l :: pat$. Alternatively, patterns may supply variables to which data values and labels in the stack are dynamically bound. For example, the pattern $x : t\ \texttt{label} :: pat$ matches stacks of the form $l :: s$ when l is a t `label` and pat matches s. After the pattern match, evaluation proceeds with l bound to x.

The operational semantics for expressions related to stack analysis is presented in Figure 8. The rules rely on an auxiliary relation $s \vdash^L pat \sim \Sigma$ to judge whether stack s matches pattern pat in label context L, and if so, the variable substitutions Σ to be applied in the `stkcase` expression. The static semantics for stack-analysis expressions, given in Figure 9, similarly relies on an auxiliary relation $pat \vdash \Gamma$ to generate from stack pattern pat the variable context Γ in which the inner `stkcase` expression should be typed.

Now consider instrumenting a function f with pre- and post-labels as one might do in a translation from a higher-level language such as MinAML. Using `store` rather than an ordinary `let` to bind f's argument gives the following:

$$\lambda x\!:\!t.\pi_1\ (f_{\texttt{post}}\langle \texttt{let}\, a = (x, \mathcal{M}(f))\, \texttt{in store}\ x = \pi_1\ f_{\texttt{pre}}\langle a\rangle\ \texttt{in}\,(e, \pi_2\ a)\rangle)$$

This new translation allows a stack pattern to extract the argument passed to f. For example, one can write a piece of before advice that takes action only when g is called directly from f and f's argument is *true*.

$\{g_{\texttt{pre}}.(x, fn) \to \texttt{stkcase stack()}$
$\quad(g_{\texttt{pre}} :: g_{\texttt{post}} :: \texttt{val:bool =true::}\ f_{\texttt{post}} :: s\!:\!\texttt{stack}\ \ \to\ \ e$ `//take action`
$\quad\texttt{|}\ \_\ \to\ (x, fn))\}$ `//just continue`

The stack matches the pattern $g_{\texttt{pre}} :: g_{\texttt{post}} :: \texttt{val:bool =true::}\ f_{\texttt{post}} :: s\!:\!\texttt{stack}$ only when control is inside the precondition advice of g but before leaving the scope of f. (The tail of the stack, matched by $s\!:\!\texttt{stack}$, can be anything.) There is some subtlety here, though: Unless *all* functions have been instrumented

$\boxed{\text{C} \longmapsto \text{C'}}$

$$\langle L, A, E[\texttt{stack()}]\rangle \longmapsto \langle L, A, E[\mathcal{S}(E)]\rangle$$

$\boxed{\text{C} \longmapsto_\beta \text{C'}}$

$$\langle L, A, \texttt{store}\ x\!:\!t = v\ \texttt{in}\ e\rangle \longmapsto_\beta \langle L, A, \texttt{stored}\ v\!:\!t\ \texttt{in}\ e[v/x]\rangle$$

$$\langle L, A, \texttt{stored}\ v\!:\!t\ \texttt{in}\ v'\rangle \longmapsto_\beta \langle L, A, v'\rangle$$

$$\frac{s \vdash^L pat \sim \Sigma}{\langle L, A, \texttt{stkcase}\ s\ \texttt{(}pat \rightarrow e_2\ \texttt{|}\ \_ \rightarrow e_3\texttt{)}\rangle \longmapsto_\beta \langle L, A, e_2[\Sigma]\rangle}$$

$$\frac{s \not\vdash^L pat \sim \Sigma}{\langle L, A, \texttt{stkcase}\ s\ \texttt{(}pat \rightarrow e_2\ \texttt{|}\ \_ \rightarrow e_3\texttt{)}\rangle \longmapsto_\beta \langle L, A, e_3\rangle}$$

$\boxed{s \vdash^L pat \sim \Sigma}$

$$\frac{}{\cdot \vdash^L \cdot \sim \cdot} \qquad \frac{}{s \vdash^L x\!:\!\texttt{stack} \sim s/x}$$

$$\frac{s \vdash^L pat \sim \Sigma}{\texttt{val}\!:\!t = v :: s \vdash^L \texttt{wild} :: pat \sim \Sigma} \qquad \frac{s \vdash^L pat \sim \Sigma}{l :: s \vdash^L \texttt{wild} :: pat \sim \Sigma}$$

$$\frac{s \vdash^L pat \sim \Sigma}{\texttt{val}\!:\!t = v :: s \vdash^L \texttt{val}\!:\!t = v :: pat \sim \Sigma}$$

$$\frac{s \vdash^L pat \sim \Sigma}{\texttt{val}\!:\!t = v :: s \vdash^L \texttt{val}\!:\!t = x :: pat \sim v/x, \Sigma}$$

$$\frac{s \vdash^L pat \sim \Sigma}{l :: s \vdash^L l :: pat \sim \Sigma} \qquad \frac{l : t \in L \qquad s \vdash^L pat \sim \Sigma}{l :: s \vdash^L x\!:\!t\ \texttt{label} :: pat \sim l/x, \Sigma}$$

Figure 8: Stack Expressions: Operational Semantics

$$\boxed{\Gamma \vdash e : t}$$

$$\frac{}{\Gamma \vdash \texttt{stack()} : \texttt{stack}} \quad \frac{}{\Gamma \vdash \cdot : \texttt{stack}} \quad \frac{l \in L}{\Gamma \vdash^{L} l : \texttt{stack}}$$

$$\frac{\Gamma \vdash s_1 : \texttt{stack} \quad \Gamma \vdash s_2 : \texttt{stack}}{\Gamma \vdash s_1 :: s_2 : \texttt{stack}} \quad \frac{\Gamma \vdash v : t}{\Gamma \vdash \texttt{val}{:}t = v : \texttt{stack}}$$

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma, x : t \vdash e_2 : t'}{\Gamma \vdash \texttt{store}\; x{:}t = e_1 \;\texttt{in}\; e_2 : t'} \quad \frac{\Gamma \vdash v : t \quad \Gamma \vdash e : t'}{\Gamma \vdash \texttt{stored}\; v{:}t \;\texttt{in}\; 'e : t'}$$

$$\frac{\Gamma \vdash e_1 : \texttt{stack} \quad pat \vdash \Gamma' \quad \Gamma, \Gamma' \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \texttt{stkcase}\; e_1\; (pat \to e_2 \;\texttt{|}\; \_ \to e_3) : t}$$

$$\boxed{pat \vdash \Gamma}$$

$$\frac{}{\cdot \vdash \cdot} \quad \frac{}{x{:}\texttt{stack} \vdash x{:}\texttt{stack}} \quad \frac{pat \vdash \Gamma}{\texttt{wild} :: pat \vdash \Gamma} \quad \frac{pat \vdash \Gamma}{\texttt{val}{:}t = v :: pat \vdash \Gamma}$$

$$\frac{pat \vdash \Gamma}{l :: pat \vdash \Gamma} \quad \frac{pat \vdash \Gamma}{\texttt{val}{:}t = x :: pat \vdash x{:}t, \Gamma} \quad \frac{pat \vdash \Gamma}{x{:}t\;\texttt{label} :: pat \vdash x{:}t\;\texttt{label}, \Gamma}$$

Figure 9: Stack Expressions: Static Semantics

with pre- and post-labels, there might be calls to arbitrarily many unlabeled functions between the $f_{\texttt{post}}$ and $g_{\texttt{pre}}$. It is possible to specify the condition that f indirectly calls g (via some other function or collection of functions) by recursively traversing the run-time stack from $g_{\texttt{post}}$ until $f_{\texttt{post}}$ is found. This sort of analysis can be useful for security purposes and is illustrated next.

Suppose function f is instead instrumented in the following manner:

$$\begin{array}{ll}\lambda x{:}t.\pi_1\ (f_{\texttt{post}}\langle \texttt{store}\ \mathit{fn} = \mathcal{M}(f)\ \texttt{in} & \texttt{let}\ a = (x, \mathit{fn})\ \texttt{in} \\ & \texttt{let}\ x = \pi_1\ f_{\texttt{pre}}\langle a\rangle\ \texttt{in}\,(e, \pi_2\ a)\ \rangle)\end{array}$$

Assuming that all function declarations are translated in this way and that the core calculus has been extended with sequential expressions and recursive functions, aspects can enforce stack-inspection-like policies.

```
{$f_{\texttt{pre}}.(x, fn) \rightarrow$
  let rec inspect s = stkcase s
   (val:string = $fn'$ :: $s'$:stack  $\rightarrow$
     if enables $fn'$ $fn$ then () else inspect $s'$
   | wild :: $s'$:stack  $\rightarrow$ inspect $s'$ // ignore labels and other values
   | _  $\rightarrow$ abort())   // reached stack bottom with no enabler found
  in inspect stack(); $(x, fn)$}
```

This aspect traverses the run-time stack of function names and checks whether the current context has enabled the function f before allowing f to execute. It relies on an auxiliary function `enables:string`$\rightarrow$`string`$\rightarrow$`bool`, which determines whether the first argument (a function name) provides the capability for the second argument (another function name) to execute. The stack-inspection code can analyze all function names in the run-time stack because these names have the same `string` type. To additionally analyze all functions' run-time arguments, the core aspect calculus could be extended with polymorphic types and type analysis, as recent research has done [DWWW05].

One of the beauties of the principle of orthogonality is that proofs of many metatheorems extend easily when new features are added. This is the case with the core language's soundness when contextual analysis is added.

Lemma 18 (Inversion of Stack Typing)
The stack typing rules are invertible.

Lemma 19 (Canonical Forms, Stacks)
If $\Gamma \vdash^L v : t$ then $t =$ `stack` implies v is a stack s.

Before proving progress, we must add a new, third case to the Decomposition I Lemma for `stack()` expressions.

Lemma 20 (Decomposition I, Extended)
If $\cdot \vdash e : t$ then either

1. *e is a value v,*

2. *e can be decomposed into $E[r]$ where r is a redex that can be reduced immediately by one of the $\longmapsto_\beta$ reductions or r has the form* `return` v `to` l*,*

3. *e has the form $E[\texttt{stack()}]$.*

Theorem 21 (Progress)
If $\vdash C$ `ok` then either the configuration is finished, or there exists another configuration C' such that $C \longmapsto C'$.

Proof The proof is nearly the same as for the core calculus without contextual analysis (Theorem 11). The only difference is that Decomposition I reveals another alternative expression of the form $E[\texttt{stack()}]$. In this case, we continue to have progress since $\langle L, A, E[\texttt{stack()}]\rangle \longmapsto \langle L, A, E[\mathcal{S}(E)]\rangle$ and $\mathcal{S}(\cdot)$ is a total function on contexts. ■

Lemma 22 (Stack Lemma)
If $\cdot \vdash E : t \Rightarrow t'$ then $\cdot \vdash \mathcal{S}(E) : \texttt{stack}$.

Proof By definition of the judgment $\cdot \vdash E : t \Rightarrow t'$, we know $x : t \vdash E[x] : t'$ with $x \notin FV(E)$. By induction on the structure of E, we can conclude that $\cdot \vdash \mathcal{S}(E) : \texttt{stack}$. ■

Definition 23 (Well-typed Substitutions)
A sequence of variable substitutions Σ has type Γ, written $\vdash^L \Sigma : \Gamma$, if and only if for all $x \in \mathrm{dom}(\Gamma)$ there exists a v such that both $v/x \in \Sigma$ and $\cdot \vdash^L v : \Gamma(x)$.

Lemma 24 (Pattern Lemma)
If $pat \vdash \Gamma$ and $\cdot \vdash s : \texttt{stack}$ and $s \vdash^L pat \sim \Sigma$ then $\vdash^L \Sigma : \Gamma$.

Proof By induction on the structure of patterns, using the inversion of typing lemma. ■

Lemma 25 (Multiple Substitutions Lemma)
If $\vdash^L \Sigma : \Gamma$ and $\Gamma \vdash e : t$ then $\cdot \vdash e[\Sigma] : t$

Proof By induction on the length of the substitution sequence Σ using the standard substitution lemma. ■

Theorem 26 (β-Preservation, Extended)
If $\vdash \langle L, A, e\rangle$ `ok` and $\langle L, A, e\rangle \longmapsto_\beta \langle L', A', e'\rangle$ then L' extends L and there is a derivation of $\vdash \langle L', A', e'\rangle$ `ok`.

Proof Our new preservation lemma extends the previous lemma. The only challenging case concerns execution of the `stkcase` operation when the first branch is taken. Here, the operational rule is:

$$\frac{(1) \quad s \vdash^L pat \sim \Sigma}{\langle L, A, \texttt{stkcase}\, s\, \texttt{(}pat \rightarrow e_2 \texttt{ | \_} \rightarrow e_3\texttt{)}\rangle \longmapsto_\beta \langle L, A, e_2[\Sigma]\rangle}$$

Since $\vdash \langle L, A, \texttt{stkcase}\, s\, \texttt{(}pat \rightarrow e_2 \texttt{ | \_} \rightarrow e_3\texttt{)}\rangle$ `ok`, we have

(2) for all $a \in A$, $\cdot \vdash^L a : \texttt{advice}$, and

(3) $\cdot \vdash^L \texttt{stkcase}\, s\, \texttt{(}pat \rightarrow e_2 \texttt{ | \_} \rightarrow e_3\texttt{)} : t$ for some t.

From (3), and by inversion of the typing rules, we can conclude that

(4) $\cdot \vdash s : \texttt{stack}$, and

(5) $pat \vdash \Gamma'$, and

(6) $\Gamma' \vdash e_2 : t$.

From (1), (4) and (5), and by the Pattern Lemma, we can conclude that

(7) $\vdash^L \Sigma : \Gamma'$

From (6) and (7), and by the Multiple Substitutions Lemma, we can conclude that

(8) $\cdot \vdash e_2[\Sigma] : t$

(2), (3) and (8) are all we need to conclude that the configuration $\langle L, A, e_2[\Sigma]\rangle$ is well typed.

■

Now that we have an extended β-Preservation lemma, we may show full Preservation.

Theorem 27 (Preservation, Extended)
If $\vdash \langle L, A, e\rangle$ `ok` *and* $\langle L, A, e\rangle \longmapsto \langle L', A', e'\rangle$ *then* L' *extends* L *and there is a derivation of* $\vdash \langle L', A', e'\rangle$ `ok`*.*

Proof We must extend the previous proof of preservation slightly as there is an additional top-level operational rule:

$$\langle L, A, E[\texttt{stack()}]\rangle \longmapsto \langle L, A, E[\mathcal{S}(E)]\rangle$$

In this case we must prove $\vdash \langle L, A, E[\mathcal{S}(E)]\rangle$ `ok`, which follows easily from the Stack Lemma. ■

4.2.1 MinAML Extensions and Interpretation

One way of providing richer context-sensitive pointcut designators in MinAML is to add the `stack()` and `stkcase` expressions directly to MinAML (with just a little syntactic sugar so programmers do not have to deal with low-level details such as the appearance of both `pre` and `post` labels in the stack). With these modifications, source language programmers can have access to a very powerful reflection mechanism.

On the other hand, one could also add a select few stack predicates to the source, as is in AspectJ, for instance. To see how to accomplish this latter design, we sketch how MinAML can be extended with some convenient syntax for expressing common context-sensitive pointcut designators.

$$\begin{array}{rcl} \mathit{pcd} & ::= & \top \mid \mathit{pcd}_1 \mathbin{\texttt{\&}} \mathit{pcd}_2 \mid \texttt{within}(f) \mid \texttt{cflow}(f) \\ \mathit{ad} & ::= & \texttt{before}\ p(x)\ \texttt{when}\ \mathit{pcd} = e \\ & \mid & \texttt{after}\ p(x)\ \texttt{when}\ \mathit{pcd} = e \\ & \mid & \texttt{around}\ p(x)\ \texttt{when}\ \mathit{pcd} = e \\ & \mid & \texttt{around}\ p(x)\ \texttt{when}\ \mathit{pcd} = e_1; \texttt{proceed}\, y \rightarrow e_2 \end{array}$$

MinAML's new *when advice* contains pointcut designators that must be satisfied for the advice to be run. The $\top$ designator is always satisfied; $\mathit{pcd}_1 \mathbin{\texttt{\&}} \mathit{pcd}_2$ is satisfied if and only if both pcd_1 and pcd_2 are satisfied; $\texttt{within}(f)$ is satisfied if and only if the current join point appears immediately within the context of function f; and $\texttt{cflow}(f)$ is satisfied if and only if the current context includes the function f. For example, the aspect

$$\texttt{before}\ h(x)\ \texttt{when}\ \texttt{within}(g)\ \texttt{\&}\ \texttt{cflow}(f) = e$$

only gets executed when f, perhaps by calling other functions, calls g, and g directly calls h. The stack of function calls must look like $h :: g :: \mathit{anything} :: f :: \mathit{anything}$, with h at the top.

MinAML with *when advice* can be translated into the core calculus in a relatively straightforward manner. First, we define a function $\mathcal{T}(p, \mathit{pcd})$ that translates a context-sensitive pointcut designator pcd for program point p (a function name) into a function from stacks to Booleans. Figure 10 reveals the details. Now, the when advice may be translated into the core much like other advice except it calls $\mathcal{T}(p, \mathit{pcd})$ to determine whether it should execute its body or do nothing. For example, the following rule shows how to translate before advice.

$$\frac{p : (t_1, t_2) \in P \quad P; \Gamma, x : t_1, \mathit{fn} : \texttt{string} \vdash e : t_1 \overset{\mathsf{term}}{\Longrightarrow} e'}{\begin{array}{l} P; \Gamma \vdash \texttt{before}\ p(x, \mathit{fn})\ \texttt{when}\ \mathit{pcd} = e \overset{\mathsf{adv}}{\Longrightarrow} \\ \quad \{p_{\texttt{pre}}.x \rightarrow \begin{array}{l} \texttt{let}\ (x, \mathit{fn}) = x\ \texttt{in} \\ \texttt{if}\ (\mathcal{T}(p, \mathit{pcd})\ \texttt{stack()})\, \texttt{then}(e', \mathit{fn})\texttt{else}(x, \mathit{fn}) \end{array} \} \end{array}}$$

The translations for the other types of advice are similar.

```
𝒯(f,⊤) = λs:stack. true

𝒯(f,p&q) = λs:stack. if (𝒯(f,p) s) then (𝒯(f,q) s) else false

𝒯(f,within(g)) =
λs:stack. stkcase s (
     f_pre :: f_post :: g_post ::s':stack → true // begin f within g
   | f_post :: g_post ::s':stack        → true // finish f within g
   | _                                  →  false)

𝒯(f,cflow(g)) =
λs:stack.
  let rec walk s' = stkcase s' (
       g_post ::s'':stack → true    // inside g
     | wild::s'':stack  → walk s''
     | _                → false)
  in walk s
```

Figure 10: Context-sensitive Pointcut Designator Translation

5 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 [KHH+01], Hyper/J [OT00], JBoss AOP [JBo05], Spring AOP [JHA+05], and dynaop [Lee05]. The apparent importance of this new programming paradigm has caused many researchers to begin to look at the semantics of aspects. The two main elements of our work that set it apart from most other efforts to give semantics to aspect-oriented languages are the fact that (1) our core calculus is typed and we believe that we were the first to develop and prove the safety of a minimal calculus of aspects and (2) that we define the semantics of an oblivious source language through a type-preserving translation into our core calculus.

Most closely related to this paper is Tucker and Krishnamurthi's work on encoding aspects in Scheme [TK03]. Their approach uses *continuation marks*, a construct introduced by Clements et al. to aid in the implementation of program debugging tools [CFF01]. 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. Related work by Masuhara, Kiczales and Dutchyn [MKD02] specifies the semantics of an aspect-oriented language in Scheme and applies partial evaluation to compile and optimize it.

Several other authors have developed small, untyped formal calculi for reasoning about aspects. For instance, Wand, Kiczales and Dutchyn [WKD02] have developed a denotational semantics for pointcuts and advice in a small aspect calculus. Jagadeesen, Jeffrey and Riely [JJR03b] develop an object-oriented, aspect-oriented language and give a specification and correctness proof for weaving. Each of these formal studies have their strengths: Wand et al. use their semantics, which is denotational (whereas the other groups are operational), to analyze some of the corner cases in the behavior of AspectJ. Jagadeesen et al.'s work sheds greater light on implementation efforts as they investigate weaving. In each case, advice and join points are directly linked to the semantics of method calls rather than being developed as an orthogonal programming constructs with their own independent semantics and neither of these works are typed.

Clifton, Leavens and Wand [CLW03] develop an untyped aspect calculus inspired by Abadi and Cardelli's object calculus. Clifton et al. focus on a *direct* study of aspects. As we discussed in the introduction, our indirect approach, in which we compile from a source language to a target language and then give semantics to the target, may make it difficult to reason about certain source-level properties; Clifton avoids this potential problem. However, we also argued that our indirect approach can help modularize and simplify the semantics of an aspect language. The complexity of the direct approach is perhaps partially revealed in the fact that Clifton's calculus has eight different syntactic classes for terms, and, in our opinion, the operational semantics they give is quite a bit more complex than ours.

More recently, Bruns et al. [BJJR04] have developed a minimal untyped calculus called μABC in which all computation is achieved through a primitive aspect mechanism. They show how to compile our MinAML into their calculus and then demonstrate how to compile μABC into the π-calculus. This research makes an interesting connection between aspect-oriented languages and concurrency theory. There are some loose connections between the semantics of our core calculus and μABC as μABC is based on manipulation of abstract "names," which are somewhat similar to our labels, and μABC has its own orthogonal mechanism for advice. However, μABC is untyped and it is unclear what sort of type theory would be needed to establish that Bruns' translations are type-preserving.

Alternative typed theories of aspect oriented programming languages include work by Aldrich [Ald04a, Ald04b], Jagadeesen, Jeffrey and Riely [JJR03a], and Clifton and Leavens [CL05]. Aldrich focuses on the interaction between aspects and modules. He develops an elegant direct semantics for aspects in the context of a calculus with simple structures and functors, and he uses logical relations to prove an interesting implementation independence property of his calculus. As in other direct-style semantics, the semantics of advice invocation is tied directly to the semantics of functions. In addition, the join point designators are somewhat impoverished in his language — he has no mechanism for implementing context-sensitive advice. Jagadeesen et al.'s work extends their earlier untyped aspect calculus with a type system. They demonstrate that typing is preserved by execution and also that weaving preserves typing. Their type system has been developed for a class-based object-oriented language and it deals with inner classes and concurrency. We have investigated none of these features in our setting so there is not much overlap in the details of the two formalisms. Clifton and Leavens focus on giving a complete semantics for around advice and proceed as it appears in AspectJ. This requires substantial machinery as they must deal with tricky issues such as advice that changes the target object of a method call. In our opinion, their semantics is much more complicated than ours, which would make it more difficult to extend the type system with advanced features such as polymorphism and effects for non-interference. On the other hand, they benefit from their efforts by obtaining a complete and accurate semantics for proceed in AspectJ.

Another typed semantics for aspects is given by Douence, Motelet and Sudholt [DMS01]. They provide a definition of pointcuts by encoding them in Haskell and they also give 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.

Lieberherr, Lorenz and Ovlinger's *Aspectual Collaborations* [LLO03, Ovl03] study the problem of how to combine aspects with modules. 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. Aspectual Collaborations enjoy a number of important properties including strong encapsulation, type safety and the possibility of separately compiling and checking module definitions. Ovlinger's thesis [Ovl03] includes a typed calculus that integrates Aspectual Collaborations with Featherweight Java, and he proves type soundness. The formalism in this work is specialized to the system of aspectual collaborations, whereas we develop a simpler, more generic platform for the study of aspects.

Bauer, Ligatti and Walker [BLW02] 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. In similar work, Douence, Fradet and Südholt [DFS02, DFS04] 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 when advice is independent of each other. It would be intriguing to know what kinds of formal laws we could prove about combinators in our aspect calculus.

More recently, Bauer et al. [BLW05] have developed a simpler semantics for their system of combinators (devoid of types and effects for noninterference), and implemented the system as an extension to Java. However, the semantics is specialized to meet the goals of explaining their system of combinators and consequently does not make an appropriate platform for experimenting with aspect-oriented design in general.

Finally, as we mentioned in the introduction, we have used the semantic framework developed in this paper to study the interactions between advice and other programming language features. In one study [DWWW05], we extended our surface language and core calculus with polymorphic pointcuts, polymorphic advice, polymorphic functions and type analysis, which are all extremely useful in many applications of aspects. In a second study [DW04], we built a type and effect system that guaranteed that aspects would not interfere with the functional behavior of the mainline program. This new type system provides programmers with the guarantee that the mainline program is not only *syntactically* oblivious to advice but also *semantically* oblivious to advice. Overall, the calculus presented in this paper provided a simple and convenient platform for studying these complex type systems.

6 Conclusions

This paper has shown that many of the features of typed 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 the simply-typed lambda calculus with aspects, and an aspect-oriented variant of the Abadi-Cardelli object calculus. We claim that this approach is relatively simple, facilitates the development of advanced type systems for aspect-oriented languages, and helps modularize proofs of important language properties such as type safety.

Acknowledgments

Many thanks to Dan Dantas, Kathleen Fisher, Stephanie Weirich, the U. Penn. PL Club and the anonymous reviewers 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

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, 1996.

[Ald04a] Jonathan Aldrich. Open modules: A proposal for modular reasoning in aspect-oriented programming. In *Workshop on foundations of aspect-oriented languages*, March 2004.

[Ald04b] Jonathan Aldrich. Open modules: Reconciling extensibility and information hiding. In *Proceedings of the Software Engineering Properties of Languages for Aspect Technologies*, March 2004.

[Asp01] Aspect-oriented programming. In Tzilla Elrad, Robert E. Filman, and Atef Bader, editors, *Special Issue of Communications of the ACM*, volume 40(10). October 2001.

[BJJR04] Glenn Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μABC: a minimal calculus for aspect-oriented programs. In *International Conference on Concurrency Theory (CONCUR)*, pages 209–224, 2004.

[BLW02] Lujo Bauer, Jarred Ligatti, and David Walker. Types and effects for non-interfereing program monitors. In *International Symposium on Software Security*, Tokyo, Japan, November 2002.

[BLW05] Lujo Bauer, Jarred Ligatti, and David Walker. Composing security policies in Polymer. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, June 2005.

[CFF01] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming*, pages 320–334, 2001.

[CL05] Curtis Clifton and Gary T. Leavens. MiniMAO: Investigating the semantics of proceed. In *Workshop on Foundations of Aspect-Oriented Languages*, March 2005. Available as Iowa State technical report No. 05-01.

[CLW03] Curtis Clifton, Gary T. Leavens, and Mitchell Wand. Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages. Technical Report TR03-13, Iowa State University, November 2003.

[DFS02] Rémi Douence, Pascal Fradet, and Mario Südholt. Detection and resolution of aspect interactions. Rapport de recherche 4435, Inria, Avril 2002.

[DFS04] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Conference on Aspect-Oriented Software Development*, pages 141–150, March 2004.

[DMS01] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In A. Yonezawa and S. Matsuoka, editors, *Proceedings of the 3rd International Conference on Reflection 2001*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Kyoto, Japan, September 2001. Springer-Verlag.

[DW04] Daniel S. Dantas and David Walker. Harmless advice. In *Workshop on Foundations of Object-oriented Languages*, January 2004.

[DWWW05] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. Polyaml: A polymorphic aspect-oriented functional programmming language. In *International Conference on Functional Programming*, September 2005.

[FF05] Robert E. Filman and Daniel P. Friedman. *Aspect-Oriented Software Development*, chapter Aspect-Oriented Programming is Quantification and Obliviousness. Addison-Wesley, 2005.

[HL94] Robert Harper and Mark 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.

[HS98] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press, 1998.

[JBo05] Jboss Aspect Oriented Programming (AOP), 2005. http://www.jboss.org/products/aop.

[JHA$^+$05] Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, and Erwin Vervaet. Spring - Java/J2EE application framework, 2005. http://www.springframework.org/docs/reference/index.html.

[JJR03a] Radha Jagadeesan, Alan Jeffrey, and James Riely. A calculus of typed aspect-oriented programs. Unpublished manuscript., 2003.

[JJR03b] Radha Jagadeesan, Alan Jeffrey, and James Riely. A calculus of untyped aspect-oriented programs. In *European Conference on Object-Oriented Programming*, Darmstadt, Germany, July 2003.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In *European Conference on Object-oriented Programming*. Springer-Verlag, 2001.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[Lee05] Bob Lee. dynaop manual, 2005. https://dynaop.dev.java.net/nonav/release/1.0-beta/manual/index.html.

[LLO03] Karl J. Lieberherr, David Lorenz, and Johan Ovlinger. Aspectual collaborations – combining modules and aspects. *The Computer Journal*, 46(5):542–565, September 2003.

[MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *ACM Conference on Functional Programming and Computer Architecture*, pages 66–77, La Jolla, June 1995.

[MKD02] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs. In Gary T. Leavens and Ron Cytron, editors, *Foundations of Aspect-Oriented Languages Workshop*, pages 17–25, April 2002.

[OT00] 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.

[Ovl03] Johan Ovlinger. *Modular Programming with Aspectual Collaborations*. PhD thesis, Northeastern University, 2003.

[TK03] David B. Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 158–167, 2003.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[WKD02] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In Gary T. Leavens and Ron Cytron, editors, *Foundations of Aspect-Oriented Languages Workshop*, pages 17–25, April 2002. Iowa State University University technical report 02-06.

[WZL03] David Walker, Steve Zdancewic, and Jarred Ligatti. A theory of aspects. In *ACM SIGPLAN International Conference on Functional Programming*, pages 127–139, Uppsala, Sweden, August 2003. ACM Press.