A Semantics for Pointcuts and Advice in Higher-Order Languages

David B. Tucker and Shriram Krishnamurthi
Department of Computer Science
Brown University
Providence, RI 02912

March 2003

Abstract

Aspect-oriented programming has proven to be a useful model for developing software that encapsulates features in separate modules. AspectJ [3], one popular aspect-oriented language, extends Java with pointcuts and advice, which allow the programmer to modify the execution of existing code. However, formal semantics for pointcuts and advice only cover first-order procedural languages. We wish to port this model of aspect-oriented programming to higher-order languages in order to explore the synergy between the aspect-oriented and functional programming paradigms.

To this end, we define an operational semantics for pointcuts and advice in a Scheme-like language. Our formulation relies on two key ideas. First, we define aspects as first-class values, which integrates well with a functional language. Second, we allow an aspect to apply to a section of code either statically or dynamically. Our semantics reduce this distinction to the related notion of static and dynamic scope.

1 Introduction

Several researchers have proposed aspect-oriented software development as an architecture that enables the programmer to encapsulate certain features as separate units of code. We often term these features "cross-cutting"—their implementation cuts across the modular structure provided by the programming language. By extending the language with new capabilities, the programmer can define each feature in its own piece of code, called an *aspect*. AspectJ, an aspect-oriented extension to Java, allows the programmer to create aspects through mechanisms called pointcuts and advice, which we shall describe in detail. However, these mechanisms currently exist only in procedural and object-oriented languages, and formal semantics work exists only for first-order procedural languages [6]. We wish to study aspects in a functional setting.

We have two reasons for examining aspects in the context of a functional language. First, many languages support first-class higher-order functions: not only traditional functional languages such as Scheme, ML, and Haskell, but mainstream languages including Perl, Python, and Ruby. We therefore need to understand how to define aspects in the presence of first-class functions. Second, we want to know whether aspects are useful in a functional language. What parts of AOP do we need, and which can we do without? Conversely, does functional programming enhance the capabilites of AOP? In a separate paper, we show how to define pointcuts and advice as first-class values in a functional language, and demonstrate some possible uses of higher-order aspects [5].

We face several challenges in specifying a formal semantics for a functional language with pointcuts and advice. For one, we need a way to identify the function at a call site; we cannot do a simple name comparison as in a first-order language. Also, to what parts of a program's execution does an aspect apply? We allude to the notions of scope in a function language and define both static and dynamic aspects. Finally, we must provide means for examining the sequence of function calls in the dynamic extent of an expression.

The remainder of the paper is organized as follows. Section 2 reviews the pointcut and advice model of aspect-oriented programming, both in Java and our functional language. Section 3 presents our operational semantics, and explains key rules in detail. Section 4 discusses related work, and Section 5 concludes.

2 Pointcuts and Advice in Higher-Order Languages

In this section, we first review the AspectJ model of aspect-oriented programming. We then describe our extension of a functional language with pointcuts and advice.

2.1 AspectJ

Since our model of aspect-oriented programming tries to mimic the style of constructs in AspectJ, we will first discuss AspectJ's definitions of pointcuts and advice. An aspect relies on modifying a program's behavior at some points in its execution, particularly points that the programmer perhaps did not anticipate in advance. They call these *join points*, and they depend both on the underlying language as well as the aspect-oriented extension to it. AspectJ defines many possible join points for Java, including method calls, variable accesses, exception throws, and object or class initialization. We will focus on method call join points because they suffice for demonstrating the utility of AspectJ.

Each join point presents an opportunity for a feature to affect the computation. The effect might be as simple as writing some trace message to output, or as complex as replacing the next computation before it occurs. The specification of an aspect therefore has two components: the *pointcut descriptor* (or *pcd*) defines the set of join points at which the aspect should apply, and the *advice* describes what computation to perform at each applicable join point.

To illustrate these two concepts, consider a simple reader library with the call graph shown in Figure 1. The following AspectJ pcd refers to any join point where the method *readProf*() calls the method *readInfo*():

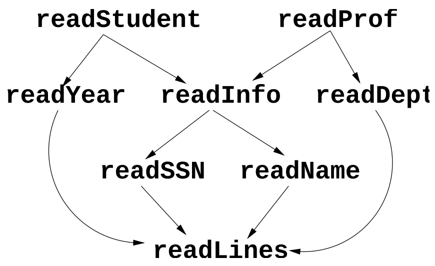

Figure 1: Call graph for a reader library

call(String *readInfo*()) *&&* **withincode**(String *readProf*())

AspectJ provides many means of defining pointcut descriptors. Simple pcd's, such as those matching method calls and variable accesses, can combine via boolean connectives to create complex pcd's, as in the above example. In addition, the construct **cflow**(p) matches any join point within the dynamic extent of a join point matching p. For example, the following pcd describes all calls to *readLines*() originating from *readStudent*():

call(String *readLines*(**int**)) *&&* **cflow**(**withincode**(String *readStudent*()))

An aspect's advice specifies what computation to perform at those join points denoted by the pcd. The programmer writes advice as standard Java code; for example, the following code prints a trace message before calling the method *readLines*():

```
before() : call(String readLines(int)) {
  System.out.println("Calling readLines");
}
```

The programmer can define different kinds of advice depending on how execution should proceed with respect to a given join point. The three basic kinds of advice are **before**, **after**, and **around**. **Before** advice executes before control enters a join point; **after** advice executes when control returns, possibly due to a thrown exception. **Around** advice replaces the current join point with a new expression to evaluate, but can reinstate the displaced computation via the keyword **proceed**. For example, the following **around** advice calls the intercepted method *readLines*(), prints the returned value, and returns the value to the original context:

```
String around() : call(String readLines(int)) {
  String s = proceed();
  System.out.println("value is " + s);
  return s;
}
```

The pcd and advice are not strictly independent entities in AspectJ. The pcd may pattern match against values in the join points it specifies; the advice can then refer to these value in its code. Aspect definitions may use this facility to capture the arguments to a method call; for example, we can verify that the argument to *readLines*() is positive before calling it:

```
String around(int n) : call(String readLines(int)) && args(n) {
  if (n <= 0)
    return "";
  else
    return proceed(n);
}
```

Some relevant points to remember about Java and AspectJ are the following:

- AspectJ defines a special language for pointcut descriptors; it includes **call** for matching a method call, and **cflow** for matching join points in the dynamic context.
- Aspects are not instantiated; they are not first-class objects.
- Methods are first-order.

In the following section, we will see how our language differs on the above points.

2.2 Defining Pointcuts and Advice

In previous work, we addressed the challenges of adding pointcuts and advice to a higher-order language [5]. Several design decisions confront us in the context of a functional language. First, how should we specify aspects? Do we create a specialized language, as in AspectJ, or make aspects first-class values? Second, since functions are inherently nameless, how do we identify them? Third, what scope does an aspect have; that is, when should a given aspect be in force? We answer these questions in this section.

We first address the specification of pointcuts and advice. We decided to make pointcuts and advice first-class values, thus enabling the programmer to use the full power of higher-order functions when manipulating aspects. Specifically, we define a pcd as a predicate over list of join points, and advice as a join point (procedure) transformer.

Consider the pointcut we saw earlier: the set of join points representing calls to *read-info* from *read-prof*. In our model, a pcd consumes a list of functions—one for each join point, with the callee as the first element, the caller as the second, and so forth—and returns true or false. Using **eq?** to compare functions for equality, we can define the pcd as follows:

```
(λ (jp)
  (and (eq? read-info (first jp))
       (not (empty? (rest jp)))
       (eq? read-prof (second jp))))
```

The primitive **eq?** deems two functions equal if they have the same source location and have identical environments. We use **eq?** to solve the problem of identifying functions in pcd's; in the above code, for example, both the variable *read-info* and the expression (*first jp*) evaluate to functions, which can then be compared using **eq?**. We will discuss the semantics in detail in the section 3.

The other pointcut we defined earlier denoted all calls to *read-lines* that originated from *read-student*. In our language, we define this pcd as follows:

```
(λ (jp)
  (and (eq? read-lines (first jp))
       (memq read-student (rest jp))))
```

where *memq* checks whether an element is a member of a list. Informally, the pcd says "return true if *read-lines* is the first join point, and *read-student* occurs in the rest of the join point list."

Since pcd's are first-class values, we can define the standard pcd operators without any special language support:

```
(call f) ≡ (λ (jp) (eq? f (first jp)))
(within f) ≡ (λ (jp) (and (not (empty? (rest jp)))
                          (eq? f (second jp))))
(cflow pcd) ≡ (λ (jp)
                (and (not (empty? jp))
                     (or (app/prim pcd jp)
                         (app/prim (app/prim cflow pcd)
                                   (rest jp))))))
(&& pcd1 pcd2) ≡ (λ (jp)
                   (and (app/prim pcd1 jp)
                        (app/prim pcd2 jp))))
```

The syntactic form **app/prim** performs a "primitive application"; that is, it applies a function to an argument without examining whether aspects apply. If we had instead defined *cflow* using (*pcd jp*), that call would itself invoke aspect weaving, which in turn would evaluate the same *cflow* pcd, leading to an infinite loop.

Using these operators, we can rewrite the two above examples as follows:

```
(&& (call read-lines) (within read-name))

(&& (call read-lines) (cflow (within read-student)))
```

Notice that these definitions look very close to the corresponding AspectJ code.

We now turn to the definition of advice. We define advice as a join point transformer: it consumes a procedure (the current join point) and returns a new procedure. For example, the following advice prints a message before invoking the original function:

```
(λ (p)
  (λ (a)
    (printf "Calling read-lines")
    (app/prim p a)))
```

In this case, the use of **app/prim** ensures that applying *p* to *a* will not invoke any further aspects, which would potentially lead to an infinite loop. This use corresponds to the AspectJ keyword **proceed**.

The other examples of advice are equally straightforward. To print the return value of a function, we write:

```
(λ (p)
  (λ (a)
    (let ([s (app/prim p a)])
      (printf "value is ~a" s)
      s)))
```

To test an argument before calling the function, we define the following advice:

```
(λ (p)
  (λ (a)
    (if (<= a 0)
        ""
        (app/prim p a))))
```

Our model has one limitation with respect to AspectJ: we can only match arguments from the current join point, whereas AspectJ can match values anywhere in the dynamic context.

Given these definitions of pointcuts and advice as first-class values, we need a mechanism for installing aspects in the program. An aspect must be able to refer to procedures in its pointcut descriptors, so it must be defined within the scope of any procedures it advises. In a first-order procedural language, there exists only one scope for procedures: the top-level scope. Thus, all aspects can also be defined in a single top-level scope. In a functional language, however, procedures may be defined at any point in the program. Therefore, we must also allow an aspect to be defined at any program point, since it needs to be in the scope of those procedures it advises. We accomplish this by adding a new expression to our language for "around" aspects:

```
(around Pcd Advice
  Body)
```

Informally, this expression means "the aspect defined by *Pcd* and *Advice* applies in *Body*." For example, we could write:

```
(let ([read-lines ...]
      [trace-advice (λ (p)
                      (λ (a)
                        (printf "Calling read-lines")
                        (app/prim p a)))])
  (around (call read-lines) trace-advice
    (list (read-lines 2)
          (read-lines 5)
          (read-lines 3)))))
```

However, we have glossed over a subtlety: what does it mean to say an aspect "applies in the body"?

To answer this question, we allude to the notions of scope in a functional language. Aspects created by (**around** *Pcd Advice Body*) are *statically* scoped. They apply to any function applications in the *text* of *Body*. Consider this example:

(**around** (*call read-lines*) *trace-advice*
  (*read-lines* 7))

In this case, the advice prints a trace message, because the application of *read-lines* to 7 occurs in the text of the body. However, an application in the text of the body does not necessarily occur while evaluating the body. Consider the following:

((**around** (*call read-lines*) *trace-advice*
   (λ (*n*) (*read-lines n*)))
  7)

The evaluation of this expression also prints a trace message, because the application of *read-lines* to *n* is in the text of **around**'s body, even though the dynamic application occurs outside.

Conversely, an **around** aspect does not apply to applications that occur during the evaluation of its body, but that were defined outside its scope. For example, the following expression *does not* print a message:

(**let** ([*apply-to-7* (λ (*f*) (*f* 7))])
  (**around** (*call read-lines*) *trace-advice*
    (*apply-to-7 read-lines*)))

Since we may wish to define aspects that *do* apply in the above example, we also allow *dynamically* scoped aspects. The expression (**fluid-around** *Pcd Advice Body*) introduces a dynamically scoped aspect: it applies to any function applications during the *evaluation* of *Body*.

If we rewrite the previous example using **fluid-around**, its evaluation *does* print a trace message:

(**let** ([*apply-to-7* (λ (*f*) (*f* 7))])
  (**fluid-around** (*call read-lines*) *trace-advice*
    (*apply-to-7 read-lines*)))

Although the application of *read-lines* occurs outside the text of **fluid-around**, it is within the dynamic extent of the **fluid-around**.

Finally, consider the case where the body of a **fluid-around** contains an application, but its evaluation does not occur during the evaluation of the body:

((**fluid-around** (*call read-lines*) *trace-advice*
   (λ (*n*) (*read-lines n*)))
  7)

This code does not print a trace message.

We will see that the definitions of static and dynamic aspects correspond to the similar notions of variable scope in a functional language, and that our semantics draw upon this similarity.

3 Semantics

In this section we lay out the semantics for a functional language extended with support for pointcuts and advice. We first give a primer on the CEKS machine, the abstraction with which we define our operational semantics. Subsequent sections then explain some key rules, including those for declaring aspects, testing function equality, and applying functions.

3.1 Background on the CEKS machine

We use a variation on the CEKS machine [2] as the model for our semantics. This model defines program behavior by a transition relation from one program state to the next. To represent the state of a program, we rely on the ability to break any expression into two pieces: the sub-expression to evaluate next, and the "rest" of the computation. We can visualize this distinction by drawing a box around the first piece; for example:

$$(+\ \mathsf{1}\ (-\ \boxed{(+\ \mathsf{2}\ \mathsf{3})}\ (*\ y\ \mathsf{4})))$$

We call the expression inside the box the *control string*; the context outside of it is the *current continuation*. In this example, the continuation says what to do with the result of $(+\ \mathsf{2}\ \mathsf{3})$:

1. next, evaluate $(*\ y\ \mathsf{4})$ and subtract it from the result
2. then, add the above result to $\mathsf{1}$
3. finally, terminate with the above result as the value of the entire computation

We can represent this continuation as a tagged list:

$$\begin{array}{l}\langle \textsf{sub-left-k}, (*\ y\ \mathsf{4}), \\ \qquad\qquad \langle \textsf{add-right-k}, \mathsf{1}, \\ \qquad\qquad\qquad\qquad \textsf{mt-k}\rangle\rangle\end{array}$$

The CEKS machine adds two more pieces of information to the state of a computation. First, it pairs each control string with an environment that maps variable names to locations in an abstract store. Second, each state has an abstract store that maps locations to value–environment pairs. Formally, we represent the state of a computation with a triple of the following form:

1. The control string (C) and its environment (E).
2. The current continuation (K).
3. The current store (S).

For example, if the above expression was inside a context where y was set to $\mathsf{5}$, the triple would be:

$$\begin{array}{l}\langle\, \langle (+\ \mathsf{2}\ \mathsf{3}), \{y \mapsto \ell_{17}\}\rangle, \\ \quad \langle \textsf{sub-left-k}, (*\ y\ \mathsf{4}), \langle \textsf{add-right-k}, \mathsf{1}, \textsf{mt-k}\rangle\rangle, \\ \quad \{\ell_{17} \mapsto \mathsf{5}\}\rangle\end{array}$$

We have three reasons for using the CEKS in defining our semantics. First, recall that pointcuts can require knowledge of the control path that led the current point in the computation; thus, we need a concrete representation of the current continuation (the stack). Second, since the machine uses an environment to maintain variables, we can easily add a second environment to keep track of aspects in scope. Third, programmers often use side-effects in writing useful aspects (e.g. logging, tracing, error reporting); hence, we include an abstract store in our model.

3.2 Declaring aspects

To declare aspects, we added the **around** and **fluid-around** expressions to a base functional language:

(**around** *Pcd Advice Body*)

(**fluid-around** *Pcd Advice Body*)

We will first describe the semantics of **around**; the semantics of **fluid-around** are nearly identical.

When the programmer declares an aspect via **around**, the machine may later access the aspect during function application. This situation resembles the use of variables: the programmer *declares* them with λ or **let**, and later *accesses* them by variable references. Drawing on this analogy, we add a second environment to our machine—one for storing aspects. The reduction rules for our model will be similar to those for the CEKS machine, except that closures now include both a variable environment and an aspect environment. The template for a reduction rule now includes *aspect environments* A_1 and A_2:

$$\langle\langle C_1, E_1, A_1\rangle, K_1, S_1\rangle \longrightarrow \langle\langle C_2, E_2, A_2\rangle, K_2, S_2\rangle$$

The evaluation of **around** has three reduction rules. The first rule moves evaluation to the pcd, M_1, while remembering that the declaration was for a `static` aspect:

$$\begin{aligned}&\langle\langle(\textbf{around}\ M_1\ M_2\ M_3), E, A\rangle, K, S\rangle\\ &\longrightarrow \langle\langle M_1, E, A\rangle, \langle\textsf{around1-k}, \texttt{static}, \langle M_2, E, A\rangle, \langle M_3, E, A\rangle, K\rangle, S\rangle\end{aligned}$$

The second rule says that once the pcd computes to a value (VC_1), evaluate the advice (MC_2) next:

$$\begin{aligned}&\langle VC_1, \langle\textsf{around1-k}, scope, MC_2, MC_3, K\rangle, S\rangle\\ &\longrightarrow \langle MC_2, \langle\textsf{around2-k}, scope, VC_1, MC_3, K\rangle, S\rangle\end{aligned}$$

The third rule applies after both the pcd and advice become values. The rule moves evaluation to the body of the **around** expression, but evaluates it with an extended aspect environment. We add the triple $\langle scope, VC_1, VC_2\rangle$ to the environment; that is, the scope tag (for **around**, it's `static`), the pcd value (VC_1), and the advice value (VC_2):

$$\begin{aligned}&\langle VC_2, \langle\textsf{around2-k}, scope, VC_1, \langle M_3, E_3, A_3\rangle, K\rangle, S\rangle\\ &\longrightarrow \langle\langle M_3, E_3, A_3[\langle scope, VC_1, VC_2\rangle]\rangle, K, S\rangle\end{aligned}$$

To support **fluid-around**, we simply add a rule similar to the first one for **around**, except that its tag is `dynamic`:

$$\langle\langle(\textbf{fluid-around}\ M_1\ M_2\ M_3), E, A\rangle, K, S\rangle$$
$$\longrightarrow \langle\langle M_1, E, A\rangle, \langle \textsf{around1-k}, \texttt{dynamic}, \langle M_2, E, A\rangle, \langle M_3, E, A\rangle, K\rangle, S\rangle$$

In short, the semantics of aspect declaration say to evaluate the pcd and advice, then add them (along with the appropriate $scope$ tag) to the aspect environment when evaluating the body.

3.3 Function equality

Next we address the issue of function identity in a higher-order language. Recall that the pointcut descriptor of an aspect can refer to one or more procedures; for example, the pcd (*call read-lines*) denotes join points representing calls to the function *read-lines*. Thus, at each function application, we must determine whether the function being applied is *read-lines*. In a language like Java, this would be an easy test—we just use string equality to compare the name *read-lines* with the name of the method being invoked. In a functional language, however, two problems arise. First, the term in the function position need not be a variable name—it may be an arbitrary expression that computes to a function. Second, even if the function *is* the variable *read-lines*, we cannot tell by its name whether this was the *read-lines* in scope when the aspect was defined. Consider the following expression:

```
(let ([read-lines ...])
  (around (call read-lines) trace-advice
    (let ([read-lines (λ (x) x)])
      (read-lines 12))))
```

In this example, should the call to *read-lines* invoke the aspect? The answer is no—because the *read-lines* in the pcd really refers to the outer *read-lines*, while the function application refers to the inner *read-lines*.

To cope with this challenge of function equality, we will borrow the definition of equality used in Scheme. The predicate **eq?** in Scheme can be used to compare functions. One interpretation of function **eq?**-ness is:

> Two function closures are equal if they have the same textual source location and their environments are identical.

To capture this meaning, we assume that each λ-expression in the source program is labelled with a unique location identifier. Two function closures are then **eq?** if and only if both identifiers are the same, and both environments are equal. The following rule illustrates this definition:

$$\langle\langle(\lambda\ (x')\ M')_{t'}, E', A'\rangle, \langle \textsf{eq2-k}, \langle(\lambda\ (x)\ M)_t, E, A\rangle, K\rangle, S\rangle$$
$$\longrightarrow \langle\langle b, \emptyset, A\rangle, K, S\rangle$$

where b is $\textsf{true}$ if $t = t'$ and $E = E'$, $\textsf{false}$ otherwise

This definition of function equality is a conservative approximation of two functions' observational equivalence, but we can compute it in constant time.

3.4 Primitive function application

Our language has two constructs for function application: the default one, which injects aspects into the computation, and a "primitive" application (named **app/prim**), which does not observe aspects. As we saw earlier, we use **app/prim** mainly to model AspectJ's **proceed** calls from within the body of an aspect's advice.

The semantics of **app/prim** are the same as that of application in the original CEKS machine, save for the question of how to handle the aspect environment. With regular (variable) environments, we have two choices:

1. We can use static scoping—we evaluate the body of the procedure using the environment from its *definition site*.
2. We can use dynamic scoping—we evaluate the body of the procedure using the environment from its *application site*.

Since we support both static and dynamic aspects, we use some aspects from both aspect environments. Specifically, we evaluate the body of the function using *static* aspects from the site of definition, and *dynamic* aspect frmo the site of appication.

The evaluation of primitive application comprises three reduction rules. The first rule moves evaluation to the function position, M_1, and keeps track of the aspect environment from the application site:

$\langle\langle(\textbf{app/prim}\ M_1\ M_2), E, A\rangle, K, S\rangle$
$\longrightarrow \langle\langle M_1, E, A\rangle, \langle \mathsf{appprim1\text{-}k}, \langle M_2, E, A\rangle, A, K\rangle, S\rangle$

The second rule moves evaluation to the argument position:

$\langle \mathit{VC}_{\mathrm{fun}}, \langle \mathsf{appprim1\text{-}k}, \mathit{MC}_{\mathrm{arg}}, A_{\mathrm{app}}, K\rangle, S\rangle$
$\longrightarrow \langle \mathit{MC}_{\mathrm{arg}}, \langle \mathsf{appprim2\text{-}k}, \mathit{VC}_{\mathrm{fun}}, A_{\mathrm{app}}, K\rangle, S\rangle$

The third rule performs the actual application. It moves evaluate to the body of the λ expression, extends the environment and store to reflect the parameter binding, and combines the two aspect environments as described above:

$\langle \mathit{VC}_{\mathrm{arg}}, \langle \mathsf{appprim2\text{-}k}, \langle(\lambda\ (x)\ M)_t, E_{\mathrm{fun}}, A_{\mathrm{fun}}\rangle, A_{\mathrm{app}}, K\rangle, S\rangle$
$\longrightarrow \langle\langle M, E', A_{\mathrm{app}}|_{\mathtt{dynamic}} \cup A_{\mathrm{fun}}|_{\mathtt{static}}\rangle, K, S'\rangle$

where $\langle E', S'\rangle = \langle E, S\rangle + \{x \mapsto \mathit{VC}_{\mathrm{arg}}\}$

To extend an environment and store with a variable and value, we use the following definition:

$$\langle E, S\rangle + \{x \mapsto \mathit{VC}\} \triangleq \langle E[x \mapsto \ell], S[\ell \mapsto \mathit{VC}]\rangle \text{ where } \ell \notin \mathrm{dom}(S)$$

$$\langle E, S\rangle + \{x_1 \mapsto \mathit{VC}_1, \ldots, x_n \mapsto \mathit{VC}_n\} \triangleq \langle E, S\rangle + \{x_1 \mapsto \mathit{VC}_1\} + \cdots + \{x_n \mapsto \mathit{VC}_n\}$$

These reduction rules for primitive application only differ from the original CEKS machine in one respect: they create the appropriate aspect environment before evaluating the body of the function.

3.5 Regular function application

We now come to the heart of our semantics: the mechanism for invoking aspects during function application.

Three things must happen during function application. First, we must generate a join point representing the application. We will do this by adding a special "application mark" (mark-app-k) to the currrent continuation which stores the callee. This mark has no direct effect on the computation; when the function returns, the following rule discards the mark:

$$\langle VC, \langle \textsf{markapp-k}, VC_{\mathrm{fun}}, K\rangle, S\rangle \longrightarrow \langle VC, K, S\rangle$$

Second, we must compute the list of current joinpoints, which each pcd takes as an argument. Since the current continuation represents each join point with exactly one mark-app-k, we compute this list via the following function:

$$\begin{array}{rcl} J[\![\textsf{mt-k}]\!] & = & \langle \textsf{empty}, \emptyset, \emptyset\rangle \\ J[\![\langle \textsf{markapp-k}, VC_{\mathrm{fun}}, K\rangle]\!] & = & \langle (\mathit{cons}\ VC_{\mathrm{fun}}\ J[\![K]\!]), \emptyset, \emptyset\rangle \\ J[\![\langle ..., K\rangle]\!] & = & J[\![K]\!] \end{array}$$

Finally, we must check each aspect in the aspect environment: if the pcd holds, we apply the advice (a procedure transformer) to the procedure. We then take the procedure resulting from all such transformations and apply it to the original argument.

Three rules dictate the evaluation of function application. The first rule moves evaluation to the function position, remembering the aspect environment from the application site:

$$\begin{array}{l} \langle\langle (M_1\ M_2), E, A\rangle, K, S\rangle \\ \longrightarrow \langle\langle M_1, E, A\rangle, \langle \textsf{app1-k}, \langle M_2, E, A\rangle, E, A, K\rangle, S\rangle \end{array}$$

The second rule moves evaluation to the argument position: eval's arg position:

$$\begin{array}{l} \langle VC_1, \langle \textsf{app1-k}, MC_2, E_{\mathrm{app}}, A_{\mathrm{app}}, K\rangle, S\rangle \\ \longrightarrow \langle MC_2, \langle \textsf{app2-k}, VC_1, E_{\mathrm{app}}, A_{\mathrm{app}}, K\rangle, S\rangle \end{array}$$

The third rule applies aspects as described above:

$$\begin{array}{l} \langle VC_{\mathrm{arg}}, \langle \textsf{app2-k}, \langle (\lambda\ (x)\ M)_t, E_{\mathrm{fun}}, A_{\mathrm{fun}}\rangle, E_{\mathrm{app}}, A_{\mathrm{app}}, K\rangle, S\rangle \\ \longrightarrow \langle\langle (\textbf{app/prim}\ W[\![|A_{\mathrm{app}}|]\!]\ \mathit{arg}), E', A_{\mathrm{app}}\rangle, K', S'\rangle \end{array}$$

where:

$$K' \quad = \quad \langle \textsf{markapp-k}, \langle (\lambda\ (x)\ M)_t, E_{\mathrm{fun}}, A_{\mathrm{fun}}\rangle, K\rangle$$

$$
\begin{array}{rcl}
\langle E', S' \rangle & = & \langle E, S \rangle \\
& + & \{ \mathit{fun} \mapsto \langle (\lambda\ (x)\ M)_t, E_{\text{fun}}, A_{\text{fun}} \rangle,\ \mathit{arg} \mapsto \mathit{VC}_{\text{arg}},\ \mathit{jp} \mapsto J[\![K']\!] \} \\
& + & \displaystyle\bigcup_{\langle scope, VC^i_{\text{pcd}}, VC^i_{\text{advice}} \rangle \in A_{\text{app}}} \{ \mathit{pcd}^i \mapsto \mathit{VC}^i_{\text{pcd}},\ \mathit{advice}^i \mapsto \mathit{VC}^i_{\text{advice}} \}
\end{array}
$$

Let us explain this rule in detail. First, it adds a mark to the continuation, creating K'. Second, it binds variables for the function argument, joinpoint list, and aspect components (the $VCpcd_i$ and $VCadvice_i$) in the environment and store. The function W then creates a new expression that checks each aspect's pcd and applies its advice if required. We define W as:

$$
\begin{array}{rcl}
W[\![0]\!] & = & \mathit{fun} \\
W[\![i]\!] & = & (\textbf{app/prim}\ (\lambda\ (r)\ (\textbf{if}\ (\textbf{app/prim}\ \mathit{pcd}^i\ \mathit{jp}) \\
& & \qquad\qquad\qquad\qquad (\lambda\ (v)\ ((\textbf{app/prim}\ \mathit{advice}^i\ r)\ v)) \\
& & \qquad\qquad\qquad\qquad r)) \\
& & \qquad\quad W[\![i-1]\!])
\end{array}
$$

Notice the following points about W:

1. If no advice exists, it simply returns the original function (to be **app/prim**'d to the argument).

2. It applies each pcd to the list of joinpoints.

3. If some pcd holds, the it applies the final transformed procedure to the original argument. Note that this application may also invoke aspects.

This concludes our discussion of the operational semantics; we present the entire set of definitions and reduction rules in the Appendix.

4 Related Work

AspectJ [3], an aspect-oriented extension to Java, allows the programmer to define pointcuts and advice. This model served as the starting point for our investigation. AspectJ uses a special language for defining pointcuts, and includes the cflow operator. The language has a tighter integration between pointcuts and advice due to matching: advice can reference data from any join point in the dynamic context, whereas our semantics limits inspection to the current join point. However, AspectJ does not address the issue of higher-order functions, and thus lacks the distinction between static and dynamic advice that we support.

Wand et al. defined a denotational semantics for AspectJ-style pointcuts and advice [6]. Their semantics defines a language with first-order procedures in a top-level recursive environment. They also use an aspect environment, but like the procedure environment, it exists in a top-level scope. Their representation of advice as a procedure transformer inspired our definition of first-class advice for a functional language.

Orleans defined a predicate dispatching system for Scheme, where each the system dispatches each message according to a *branch* consisting of a predicate and a body [4]. These two components are first class values, and thus equivalent to our pcd and advice formulation. His system also supports **cflow** by having each decision point (join point) maintain a pointer to the previous decision point. Our system differs from his in two ways. First, we do not require the programmer to use a special message syntax. Second, Orleans does not address the issue of scope—in his system, the programmer must define messages at the top level.

In previous work, we argue for the usefulness of our aspect-oriented language, and describe a lightweight implementation in PLT Scheme [5]. We give examples that illustrate the power of higher-order pointcuts and advice, and exploit the distinction between static and dynamic aspects. Our implementation relies on two features of PLT Scheme: the ability to define new languages using macros and the module system, and a language feature ("continuation marks" [1]) that allow us to use a built-in dynamic environment.

5 Conclusion

We have presented an operational semantics for pointcuts and advice in a higher-order language. We based our semantics on the CEKS machine, which represents the current continuation explicitly as a list, and uses an abstract store to model state. Our work has two key design points. First, we decided to define pointcuts and advice as first-class values: a pointcut is a predicate over a list of join points, and advice is a procedure transformer. Second, we distinguish between static aspects, which apply to the text of an expression no matter where it executes, and dynamic aspects, which apply to computations during the evaluation of an expression. The semantics implement this behavior by maintaining an aspect environment, and employing it in accordance with the standard rules for static and dynamic scope.

References

[1] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. *Lecture Notes in Computer Science*, 2028, 2001.

[2] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.

[3] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, 2001.

[4] Doug Orleans. Incremental programming with extensible decisions. In *International Conference on Aspect-Oriented Software Development*, 2002.

[5] David B. Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In *International Conference on Aspect-Oriented Software Development*, 2003.

[6] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. Appeared in Informal Workshop Record of Foundations of Object-Oriented Languages 9, pages 67-88, 2002.

Appendix

Expressions

$$
\begin{array}{lcl}
M & ::= & x \\
 & | & (\lambda\ (x)\ M)_t \\
 & | & (M\ M) \\
 & | & (o\ M\ \ldots) \\
 & | & (\textbf{if}\ M\ M\ M) \\
 & | & (\textbf{set!}\ x\ M) \\
 & | & (\textbf{eq?}\ M\ M) \\
 & | & (\textbf{around}\ M\ M\ M) \\
 & | & (\textbf{fluid-around}\ M\ M\ M) \\
 & | & (\textbf{app/prim}\ M\ M)
\end{array}
$$

t = source location tag

Primitive operations

$$
\begin{array}{lcl}
o & ::= & \mathit{cons} \\
 & | & \mathit{first} \\
 & | & \mathit{rest} \\
 & | & \mathit{empty?}
\end{array}
$$

Environments $E = x \mapsto \ell$

ℓ = store location

Aspect environments $A = \{\langle \mathit{scope}, \mathit{VC}, \mathit{VC} \rangle\}$

$\mathit{scope} \in \{\texttt{static}, \texttt{dynamic}\}$

Stores $S = \ell \mapsto \mathit{VC}$

Expression closures $\mathit{MC} = \langle M, E, A \rangle$

Value closures $\mathit{VC} = \langle V, E, A \rangle$

Values

$$
\begin{array}{lcl}
V & ::= & (\lambda\ (x)\ M)_t \\
 & | & b
\end{array}
$$

$$
\begin{array}{llll}
\textbf{Primitive values} & b & ::= & \textsf{true} \\
 & & | & \textsf{false} \\
 & & | & \textsf{empty} \\
 & & | & (\mathit{cons}\ \mathit{VC}\ \mathit{VC}) \\
 & & & \\
\textbf{Continuation codes} & K & ::= & \textsf{mt-k} \\
 & & | & \langle \textsf{app1-k}, \mathit{MC}, E, A, K \rangle \\
 & & | & \langle \textsf{app2-k}, \mathit{VC}, E, A, K \rangle \\
 & & | & \langle \textsf{if-k}, \mathit{MC}, \mathit{MC}, K \rangle \\
 & & | & \langle \textsf{set-k}, \langle x, E, A \rangle, K \rangle \\
 & & | & \langle \textsf{eq1-k}, \mathit{MC}, K \rangle \\
 & & | & \langle \textsf{eq2-k}, \mathit{VC}, K \rangle \\
 & & | & \langle \textsf{around1-k}, \mathit{scope}, \mathit{MC}, \mathit{MC}, K \rangle \\
 & & | & \langle \textsf{around2-k}, \mathit{scope}, \mathit{VC}, \mathit{MC}, K \rangle \\
 & & | & \langle \textsf{markapp-k}, \mathit{VC}, K \rangle \\
 & & | & \langle \textsf{appprim1-k}, \mathit{MC}, A, K \rangle \\
 & & | & \langle \textsf{appprim2-k}, \mathit{VC}, A, K \rangle \\
 & & | & \langle \textsf{op-k}, o, \langle \mathit{VC}, ... \rangle, \langle \mathit{MC}, ... \rangle, K \rangle
\end{array}
$$

variables

$$\langle \langle x, E, A \rangle, K, S \rangle \longrightarrow \langle S(E(x)), K, S \rangle$$

function application

$$\langle \langle (M_1\ M_2), E, A \rangle, K, S \rangle$$
$$\longrightarrow \langle \langle M_1, E, A \rangle, \langle \textsf{app1-k}, \langle M_2, E, A \rangle, E, A, K \rangle, S \rangle$$

$$\langle \mathit{VC}_1, \langle \textsf{app1-k}, \mathit{MC}_2, E_{\mathrm{app}}, A_{\mathrm{app}}, K \rangle, S \rangle$$
$$\longrightarrow \langle \mathit{MC}_2, \langle \textsf{app2-k}, \mathit{VC}_1, E_{\mathrm{app}}, A_{\mathrm{app}}, K \rangle, S \rangle$$

$$\langle \mathit{VC}_{\mathrm{arg}}, \langle \textsf{app2-k}, \langle (\lambda\ (x)\ M)_t, E_{\mathrm{fun}}, A_{\mathrm{fun}} \rangle, E_{\mathrm{app}}, A_{\mathrm{app}}, K \rangle, S \rangle$$
$$\longrightarrow \langle \langle (\textbf{app/prim}\ W[\![|A_{\mathrm{app}}|]\!]\ \mathit{arg}), E', A_{\mathrm{app}} \rangle, K', S' \rangle$$

where:

$$K' = \langle \textsf{markapp-k}, \langle (\lambda\ (x)\ M)_t, E_{\mathrm{fun}}, A_{\mathrm{fun}} \rangle, K \rangle$$

$$
\begin{array}{rcl}
\langle E', S' \rangle & = & \langle E, S \rangle \\
 & + & \{\mathit{fun} \mapsto \langle (\lambda\ (x)\ M)_t, E_{\mathrm{fun}}, A_{\mathrm{fun}} \rangle,\ \mathit{arg} \mapsto \mathit{VC}_{\mathrm{arg}},\ \mathit{jp} \mapsto J[\![K']\!]\} \\
 & + & \displaystyle\bigcup_{\langle \mathit{scope}, \mathit{VC}^i_{\mathrm{pcd}}, \mathit{VC}^i_{\mathrm{advice}} \rangle \in A_{\mathrm{app}}} \{\mathit{pcd}^i \mapsto \mathit{VC}^i_{\mathrm{pcd}},\ \mathit{advice}^i \mapsto \mathit{VC}^i_{\mathrm{advice}}\}
\end{array}
$$

$$
\begin{array}{rcl}
W[\![0]\!] & = & \mathit{fun} \\
W[\![i]\!] & = & (\textbf{app/prim}\ (\lambda\ (r)\ (\textbf{if}\ (\textbf{app/prim}\ pcd^i\ \mathit{jp}) \\
 & & \qquad\qquad (\lambda\ (v)\ ((\textbf{app/prim}\ advice^i\ r)\ v)) \\
 & & \qquad\qquad r)) \\
 & & \quad W[\![i-1]\!])
\end{array}
$$

$$
\begin{array}{rcl}
J[\![\textsf{mt-k}]\!] & = & \langle \textsf{empty}, \emptyset, \emptyset\rangle \\
J[\![\langle \textsf{markapp-k}, \mathit{VC}_{\text{fun}}, K\rangle]\!] & = & \langle (\mathit{cons}\ \mathit{VC}_{\text{fun}}\ J[\![K]\!]), \emptyset, \emptyset\rangle \\
J[\![\langle ..., K\rangle]\!] & = & J[\![K]\!]
\end{array}
$$

$$
\begin{array}{rcl}
\langle E, S\rangle + \{x \mapsto \mathit{VC}\} & \triangleq & \langle E[x \mapsto \ell], S[\ell \mapsto \mathit{VC}]\rangle \text{ where } \ell \notin \mathrm{dom}(S) \\
\langle E, S\rangle + \{x_1 \mapsto \mathit{VC}_1, \ldots, x_n \mapsto \mathit{VC}_n\} & \triangleq & \langle E, S\rangle + \{x_1 \mapsto \mathit{VC}_1\} + \cdots + \{x_n \mapsto \mathit{VC}_n\}
\end{array}
$$

$\langle \mathit{VC}, \langle \textsf{markapp-k}, \mathit{VC}_{\text{fun}}, K\rangle, S\rangle \longrightarrow \langle \mathit{VC}, K, S\rangle$

primitive operations

$\langle\langle (o\ M_1\ M_2\ \ldots), E, A\rangle, K, S\rangle$
$\longrightarrow \langle\langle M_1, E, A\rangle, \langle \textsf{op-k}, o, \langle\rangle, \langle\langle M_2, E, A\rangle, ...\rangle, K\rangle, S\rangle$

$\langle \mathit{VC}, \langle \textsf{op-k}, o, \langle \mathit{VC}', ...\rangle, \langle \mathit{MC}, \mathit{MC}', ...\rangle, K\rangle, S\rangle$
$\longrightarrow \langle \mathit{VC}', \langle \textsf{op-k}, o, \langle \mathit{VC}, \mathit{VC}', ...\rangle, \langle \mathit{MC}', ...\rangle, K\rangle, S\rangle$

$\langle \mathit{VC}_n, \langle \textsf{op-k}, o, \langle \mathit{VC}_{n-1}, ..., \mathit{VC}_1\rangle, \langle\rangle, K\rangle, S\rangle$
$\longrightarrow \langle \delta(o, \mathit{VC}_1, ..., \mathit{VC}_n), K, S\rangle$

$$
\begin{array}{rcl}
\delta(\mathit{cons}, \mathit{VC}_1, \mathit{VC}_2) & = & (\mathit{cons}\ \mathit{VC}_1\ \mathit{VC}_2) \\
\delta(\mathit{first}, (\mathit{cons}\ \mathit{VC}_1\ \mathit{VC}_2)) & = & \mathit{VC}_1 \\
\delta(\mathit{rest}, (\mathit{cons}\ \mathit{VC}_1\ \mathit{VC}_2)) & = & \mathit{VC}_2 \\
\delta(\mathit{empty?}, \mathit{VC}) & = & \langle \textsf{true}, \langle\emptyset, u\rangle, \emptyset\rangle \text{ if } \mathit{VC} = \langle \textsf{empty}, E, A\rangle, u \text{ fresh} \\
 & = & \langle \textsf{false}, \langle\emptyset, u\rangle, \emptyset\rangle \text{ otherwise}, u \text{ fresh}
\end{array}
$$

if

$\langle\langle (\textbf{if}\ M_1\ M_2\ M_3), E, A\rangle, K, S\rangle$
$\longrightarrow \langle\langle M_1, E, A\rangle, \langle \textsf{if-k}, \langle M_2, E, A\rangle, \langle M_3, E, A\rangle, K\rangle, S\rangle$

$\langle\langle \textsf{true}, E, A\rangle, \langle \textsf{if-k}, \mathit{MC}_2, \mathit{MC}_3, K\rangle, S\rangle$
$\longrightarrow \langle \mathit{MC}_2, K, S\rangle$

$\langle\langle \mathsf{false}, E, A\rangle, \langle \mathsf{if\text{-}k}, \mathit{MC}_2, \mathit{MC}_3, K\rangle, S\rangle$
$\longrightarrow \langle \mathit{MC}_3, K, S\rangle$

set!

$\langle\langle (\mathbf{set!}\ x\ M_1), E, A\rangle, K, S\rangle$
$\longrightarrow \langle\langle M_1, E, A\rangle, \langle \mathsf{set\text{-}k}, \langle x, E, A\rangle, K\rangle, S\rangle$

$\langle \mathit{VC}, \langle \mathsf{set\text{-}k}, \langle x, E, A\rangle, K\rangle, S\rangle$
$\longrightarrow \langle\langle \mathit{VC}, \emptyset, \emptyset\rangle, K, S[E(x) \mapsto \mathit{VC}]\rangle$

eq?

$\langle\langle (\mathbf{eq?}\ M_1\ M_2), E, A\rangle, K, S\rangle$
$\longrightarrow \langle\langle M_1, E, A\rangle, \langle \mathsf{eq1\text{-}k}, \langle M_2, E, A\rangle, K\rangle, S\rangle$

$\langle \mathit{VC}_1, \langle \mathsf{eq1\text{-}k}, \mathit{MC}_2, K\rangle, S\rangle$
$\longrightarrow \langle \mathit{MC}_2, \langle \mathsf{eq2\text{-}k}, \mathit{VC}_1, K\rangle, S\rangle$

$\langle\langle (\lambda\ (x')\ M')_{t'}, E', A'\rangle, \langle \mathsf{eq2\text{-}k}, \langle (\lambda\ (x)\ M)_t, E, A\rangle, K\rangle, S\rangle$
$\longrightarrow \langle\langle b, \emptyset, A\rangle, K, S\rangle$

where b is true if $t = t'$ and $E = E'$, false otherwise

around and **fluid-around**

$\langle\langle (\mathbf{around}\ M_1\ M_2\ M_3), E, A\rangle, K, S\rangle$
$\longrightarrow \langle\langle M_1, E, A\rangle, \langle \mathsf{around1\text{-}k}, \mathtt{static}, \langle M_2, E, A\rangle, \langle M_3, E, A\rangle, K\rangle, S\rangle$

$\langle\langle (\mathbf{fluid\text{-}around}\ M_1\ M_2\ M_3), E, A\rangle, K, S\rangle$
$\longrightarrow \langle\langle M_1, E, A\rangle, \langle \mathsf{around1\text{-}k}, \mathtt{dynamic}, \langle M_2, E, A\rangle, \langle M_3, E, A\rangle, K\rangle, S\rangle$

$\langle \mathit{VC}_1, \langle \mathsf{around1\text{-}k}, \mathit{scope}, \mathit{MC}_2, \mathit{MC}_3, K\rangle, S\rangle$
$\longrightarrow \langle \mathit{MC}_2, \langle \mathsf{around2\text{-}k}, \mathit{scope}, \mathit{VC}_1, \mathit{MC}_3, K\rangle, S\rangle$

$\langle \mathit{VC}_2, \langle \mathsf{around2\text{-}k}, \mathit{scope}, \mathit{VC}_1, \langle M_3, E_3, A_3\rangle, K\rangle, S\rangle$
$\longrightarrow \langle\langle M_3, E_3, A_3[\langle \mathit{scope}, \mathit{VC}_1, \mathit{VC}_2\rangle]\rangle, K, S\rangle$

app/prim

$\langle\langle (\mathbf{app/prim}\ M_1\ M_2), E, A\rangle, K, S\rangle$
$\longrightarrow \langle\langle M_1, E, A\rangle, \langle \mathsf{appprim1\text{-}k}, \langle M_2, E, A\rangle, A, K\rangle, S\rangle$

$\langle \mathit{VC}_{\mathrm{fun}}, \langle \mathsf{appprim1\text{-}k}, \mathit{MC}_{\mathrm{arg}}, A_{\mathrm{app}}, K\rangle, S\rangle$

$\longrightarrow \langle MC_{\mathrm{arg}}, \langle \textsf{appprim2-k}, VC_{\mathrm{fun}}, A_{\mathrm{app}}, K \rangle, S \rangle$

$\langle VC_{\mathrm{arg}}, \langle \textsf{appprim2-k}, \langle (\lambda\ (x)\ M)_t, E_{\mathrm{fun}}, A_{\mathrm{fun}} \rangle, A_{\mathrm{app}}, K \rangle, S \rangle$
$\longrightarrow \langle \langle M, E', A_{\mathrm{app}}|_{\mathtt{dynamic}} \cup A_{\mathrm{fun}}|_{\mathtt{static}} \rangle, K, S' \rangle$

where $\langle E', S' \rangle = \langle E, S \rangle + \{x \mapsto VC_{\mathrm{arg}}\}$