# Pointcuts and Advice in Higher-Order Languages [*]

David B. Tucker and Shriram Krishnamurthi
Department of Computer Science
Brown University
{dbtucker, sk}@cs.brown.edu

## ABSTRACT

Aspect-oriented software design will need to support languages with first-class and higher-order procedures, such as Python, Perl, ML and Scheme. These language features present both challenges and benefits for aspects. On the one hand, they force the designer to carefully address issues of scope that do not arise in first-order languages. On the other hand, these distinctions of scope make it possible to define a much richer variety of policies than first-order aspect languages permit.

In this paper, we describe the subtleties of pointcuts and advice for higher-order languages, particularly Scheme. We then resolve these subtleties by alluding to traditional notions of scope. In particular, programmers can now define both dynamic aspects traditional to AOP and static aspects that can capture common security-control paradigms. We also describe the implementation of this language as an extension to Scheme. By exploiting two novel features of our Scheme system—continuation marks and language-defining macros—the implementation is lightweight and integrates well into the programmer's toolkit.

## 1. INTRODUCTION

Current programming languages offer many ways of organizing code into conceptual blocks, whether through functions, objects, modules, or some other mechanism. However, programmers often encounter features that do not correspond well to these units of organization. Such features are said to "cross cut" the design of a system, because the code that implements the feature appears across many program units. In a procedural language, such a feature might be implemented as pieces of disjoint procedures; in an object-oriented language, the feature might span several methods or even objects. These cross-cutting features inhibit software development in many ways. For one, it is difficult for the programmer to reason about how the disparate pieces of the feature interact. Also, they break modularity: the programmer cannot simply add or delete these features from a program, since they are not separable units.

[*]This research was partially supported by NSF grants ESI-0010064 and ITR-0218973.

Recently, many researchers have proposed aspect-oriented software development (AOSD) as a method for organizing cross-cutting features [12, 4, 10, 17, 2, 16, 18]. In particular, Kiczales et al. have presented aspect-oriented programming (AOP); in this paradigm, the pieces of each feature combine to form a separate component, called an *aspect*. In addition to containing the code necessary for a feature, the aspect must indicate when this code should be run during program execution. Kiczales et al. also implemented a practical aspect-oriented extension to Java, called AspectJ, which allows the programmer to define pointcuts and advice and integrates them into a program [11].

Current languages that support AOP, such as AspectJ, have been built as extensions to object-oriented and first-order procedural languages. The goal of our work is to understand the relationship between AOP and functional programming. Two reasons motivate our investigation of this topic. On one hand, there are many widely used functional languages that could benefit from AOP. In addition to conventional functional languages like ML, Scheme, and Haskell, many new languages, in particular "scripting" languages such as Perl and Python, now include anonymous and higher-order functions. As more and more functional languages emerge, we need to understand the feasibility and utility of aspect-oriented programming is these languages. On the other hand, we can ask whether the greater abstractive power of functional programming enhances AOP. We might be able to simplify the specification of aspects because the underlying language provides a stronger framework for defining linguistic extensions. Also, we might use parametricity to define more general aspects, and develop aspect combinators by employing higher-order functions. Clearly, the interaction between AOP and functional programming merits careful investigation.

The two main challenges to adding AspectJ-style aspects to a functional language are the specification of aspects and the defining the scope of an aspect's applicability. First, we need to decide how to specify pointcuts and advice. Are they new kinds of values? Do we need a sublanguage for describing when aspects are to be invoked? Second, we must address the issue of scope. Unlike a first-order language, where all procedures and aspects are declared at the top level and have global scope, definitions in a higher-order language can be introduced at any point and have more specific scope. We must decide the scope in which an aspect can affect program execution.

We will address these challenges by defining an aspect-oriented extension to a functional language. Section 2 presents background on AspectJ, and puts forth our aspect-oriented extension to Scheme. Section 3 gives examples of pointcuts and advice in our language, and discusses the synergy between AOP and functional programming. Section 4 informally presents the semantics for our extension. In Section 5, we describe in detail a lightweight implementation of aspects. Section 6 discusses related work, and section 7 concludes.

## 2. DEFINING POINTCUTS AND ADVICE

This section first describes AspectJ's model of aspect-oriented programming. We then present our formulation of AspectJ-style pointcuts and advice in a higher-order language, and then address the issue of obliviousness with respect to our system.

### 2.1 A Brief Overview of AspectJ

AspectJ allows the programmer to modify a program's behavior at certain points during its execution, called *join points*. In Java, these points include method calls, variables accesses, exception throws, and object or class initialization. We will focus on method call join points because they are sufficient for demonstrating the utility of AspectJ.

Each join point presents an opportunity for an aspect 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 has two components: the *pointcut descriptor* (or *pcd*), which determines when the aspect should apply, and the *advice*, which describes what computation to perform.

AspectJ provides several primitive pointcut descriptors; two important pcd's are **call(***m***)**, which matches calls to method *m*, and **cflow(***p***)**, which matches any join point within the dynamic extent of a join point matching *p*.

An aspect's advice specifies what computation to perform at those join points denoted by the pcd. 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 but can reinstate the displaced computation via the keyword **proceed**.

The pcd and advice are not strictly independent entities in AspectJ. The pcd may pattern match against values in the join points it specifies; these values can then be referenced in the advice code. Aspect definitions often use this facility to capture the arguments to a method call.

### 2.2 Definitions in Higher-Order Languages

We support pointcuts and advice in the presence of higher-order functions, while retaining the essential features of AspectJ. We chose PLT Scheme as the sandbox for our experimentation because it provides strong support for linguistic extensibility: higher-order functions, dynamic types, and a powerful macro system. PLT Scheme makes it particularly easy to define new languages using the macro system, a feature we will rely on for developing a lightweight implementation of aspects.

The fulcrum of our approach is that like functions, both pointcuts and advice should be first-class entities. This decision is consistent with the design of functional languages, and enables us to experiment with defining more general aspects. For example, we can define aspects that are parameterized over some set of variables, or write aspect transformers or combinators using higher-order functions; we will demonstrate several such examples. First, though, we must answer the question: what are first-class pointcuts and advice?

Recall the defintion of a pointcut descriptor: it describes a set of join points over which an aspect's advice applies. In a higher-order language, the natural way to describe a set is by an inclusion predicate. Thus, we represent a pcd as a function that consumes the list of join points in the dynamic context and returns true or false. To test whether the current join point represents a call to the function *f*, the predicate would be:

```
(lambda (jp)
  (eq? f (first jp)))
```

where *jp* is the list of dynamic join points. The first element in this list is the most recent join point, the second element is the next most recent join point we have not exited, and so forth.

This code illustrates two key points about our specification of pcd's. A pcd is first-class: it can be any Scheme expression that evaluates to a predicate over a list of join points. Also, to test whether two functions are identical, we use the built-in Scheme predicate *eq?*. This predicate tests for equality on procedures by verifying that these two conditions hold: the two procedures were defined in the same source location, and the environments stored in their closures are identical.

This definition of procedure equality differs notably from the definition used in first-order languages. In a first-order language, every procedure has a global name, and can be unambiguously identified by that name. However, in Scheme and most higher-order languages, functions are inherently nameless. Futhermore, since higher-order functions close over variables external to their definitions, a single function may be instantiated in two different environments. Clearly, these two instantiations are not equivalent, since they produce different results for a given input. Scheme's *eq?* provides a useful and fast conservative approximation of observational equivalence between functions.

Of course, a pointcut descriptor can look further than the most recent join point. To match all direct calls from *g* to *f*, we write:

```
(lambda (jp)
  (and (eq? f (first jp))
       (eq? g (second jp))))
```

Or we can match any calls to *f* within the dynamic extent of *g*:

```
(lambda (jp)
  (and (eq? f (first jp))
       (memq g (rest jp))))
```

Using this formulation, we can define the standard pcd builders *call* and *within* as follows:

```
(call f) ≡ (lambda (jp) (eq? f (first jp)))
```

```
(within f) ≡ (lambda (jp) (and (not (empty? (rest jp)))
                               (eq? f (second jp))))
```

We can also define the pcd operators *cflow* and *&&*:

```
(cflow pcd) ≡ (lambda (jp)
                (and (not (empty? jp))
                     (or (app/prim pcd jp)
                         (app/prim (app/prim cflow pcd)
                                   (rest jp))))))
(&& pcd1 pcd2) ≡ (lambda (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.

Now that we have defined pcd's as first-class values, we turn to defining advice. We will focus on **around** advice, since it is strictly more general than both **before** and **after**. We consider a slightly simpler form of **around** advice than what AspectJ allows—we view it as a transformation on the current join point. Specifically, we define advice as a function transformer that consumes the original function to be executed, and returns a function to use in its place. This formulation of advice is similar to the denotational semantics of advice given by Wand et al. [20] for a first-order procedural language. For example, we can define the following advice:

```
(lambda (p)
  (lambda (a)
    (printf "aborted call to ~s with args ~s" p a)
    17))
```

The first parameter, *p*, is the function to transform; the second parameter, *a*, is the argument passed to that function. When this advice captures the function call, it prints an error message and returns the value 17 without calling the original function. More interestingly, we can define advice that changes the argument to the captured function:

```
(lambda (p) (lambda (a) (app/prim p (+ a 83))))
```

This advice adds 83 to the argument before calling the function. We employ **app/prim** to capture the behavior of AspectJ's **proceed**, which applies the original function without performing any additional aspect weaving.

Having seen how both pcd's and advice are specified as first-class entities, we now need a way to install them in the computation. To that end, we introduce a new term in our language named **around**:

```
(around pcd advice body)
```

Informally, the semantics of this expression is to evaluate *body* under the aspect defined by *pcd* and *advice*. A simple example of its use is:

```
(define (double x) (+ x x))
(around (call double) (lambda (p)
                        (lambda (a)
                          (printf "calling double")
                          (app/prim p a)))
  (double 143))
```

When executed, this program prints a string to standard output, and returns the value 286.

While the **around** construct may ostensibly seem straightforward, we have blithely ignored a critical issue: the extent of an aspect's jurisdiction. In defining **around**, we need to be more specific about *when* the aspect will be active, especially in the presence of higher-order functions. Fortunately, the problem of reasoning about the extent is a familiar one: we encounter it in defining whether variables should be statically or dynamically scoped. In a first-class procedural value, statically-scoped variables get their values from the environment of the procedure's definition; dynamically-scoped variables get their values from the environment of the procedure's invocation.

We exploit this distinction for defining aspects also. A static aspect declaration applies to an expression no matter where it is used. If the body of the declaration is a procedure, then the aspect applies in the use of the procedure. In contrast, a dynamic aspect applies only in its dynamic extent, which is the body of the aspect declaration. When the body finishes computing, the aspect no longer applies. Any procedures defined in the body do not apply the aspect outside that extent.

The **around** construct creates static aspects; we also support a construct **fluid-around** which creates dynamic aspects. To understand the difference, consider these definitions:

```
(define (add2 x) (+ x 2))
(define trace-advice (lambda (p)
                       (lambda (a)
                         (printf "calling add2")
                         (app/prim p a))))
```

Consider each of the following uses of this advice:

```
((around (call add2) trace-advice (lambda (v) (add2 v))) 7)
```

applies the aspect statically. Therefore, the aspect is in force when the procedure is applied to 7. As a result, it prints a message before returning 9. In contrast, in the following use,

```
((fluid-around (call add2) trace-advice (lambda (v) (add2 v))) 7)
```

the extent of the **fluid-around** terminates before the procedure is applied, resulting in no console output.

Now suppose we define the following function as well:

```
(define (apply-to-4 f) (f 4))
```

Suppose we now attempt to apply *trace-advice*:

```
(around (call add2) trace-advice
  (apply-to-4 add2))
```

*apply-to-4* has no aspects present as its definition. Therefore, the advice is never invoked. In contrast, applying the advice dynamically

```
(fluid-around (call add2) trace-advice
  (apply-to-4 add2))
```

results in console output.

## 2.3 Obliviousness

Consider the following Java program:

```
public class Point {
  int x, y;
  Point (int x, int y) { this.x = x; this.y = y; }
  int getX() { return x; }
  int getY() { return y; }

  public static void main(String args[]) {
    Point p = new Point(3, 4);
    p.getX(); }}
```

By writing the following AspectJ code, we can trace all calls to *getX*():

```
public aspect Trace {
  before() : call(int getX()) { ... }}
```

This alteration has the key property that the programmer did not have to anticipate any future changes. This property is called *obliviousness* [7]; it ensures that aspects can affect the behavior of a program whose original source does not contain any references to aspects.

Consider the following transliteration of the above Java program into Scheme:

```
(define (new-point x y) ...)
(define (point-get-x pt) ...)
(define (point-get-y pt) ...)

(define (main)
  (let ([p (new-point 3 4)])
    (point-get-x p)))
```

The programmer can now separately write the following aspect:

```
(fluid-around (call point-get-x) (lambda (p) (lambda (a) ...))
  (main))
```

Notice that we were able to modify the behavior of the original program without leaving any hooks in it. Based on examples such as this, we therefore believe our version of pointcuts and advice have the same power of obliviousness as AspectJ.

The degree of obliviousness depends on the ability to name entities. Some of these entities are first-order (and named), such as classes in Java and top-level definitions in Scheme. For these, it is relatively easy to obliviously modify their behavior—both AspectJ and our system provide comparable power.

Other entities are often anonymous and first-class. Examples of these include both closures in Scheme and objects in Java. The anonymity makes it is difficult to modify their behavior primarily because it is difficult to identify them in the first place. We can nevertheless name an expression such as **new** ... in Java using static distance coordinates. However, this does not give us a way to distinguish among the potentially infinite number of objects generated at this creation site.

Some closures in Scheme are analogous to objects in Java. A procedure that is nested within another procedure is not instantiated until the outer procedure is invoked, and may result in a potentially infinite number of closures at run time. Obliviously modifying the behavior of these procedures therefore poses the exact same problems as doing it for objects in Java.

## 3. PROGRAMMING WITH ASPECTS IN SCHEME

We have seen the language features necessary for adding pointcuts and advice to a functional language. In this section, we present examples demonstrating the interaction between functional programming and aspect-oriented programming. First, we give a simple program that benefits from the use of both static and dynamic aspects. Second, we show examples of how higher-order aspects are both feasible and useful.

### 3.1 Static and Dynamic Aspects

To study the utility of aspects in a functional language, we will look at an example of how we can implement a security model using a combination of static and dynamic aspects. Consider this scenario: we want to provide a simple operating system API to an untrusted client program. This API contains three functions: *read-file*, *write-file*, and *run-program*. The original code is organized by function, with security checks scattered throughout:

```
(define (read-file f)
  (if (no-read-permission? user)
      (raise 'no-permission-exception)
      (let ([p (open-file f)])
        ...)))

(define (write-file f)
  (if (no-write-permission? user)
      (raise 'no-permission-exception)
      (let ([p (open-file f)])
        ...)))

(define (run-program p)
  (if (no-run-permission? user)
      (raise 'no-permission-exception)
      (load&run p)))

(list read-file write-file run-program) ;; export three functions
```

First, we would like to factor out the permission-checking code into aspects:

```
(define (read-file f)
  (let ([p (open-file f)])
    ...))

(define (write-file f)
  (let ([p (open-file f)])
     ...))

(define (run-program p)
  (load&run p))

(define read-pcd (&& (call open-file) (within read-file)))
(define read-adv (lambda (p) (lambda (a)
                   (if (no-read-permission? user)
                       (raise 'no-permission-exception)
                       (app/prim p a)))))
```

```
(define write-pcd (&& (call open-file) (within write-file)))
(define write-adv (lambda (p) (lambda (a)
                      (if (no-read-permission? user)
                          (raise 'no-permission-exception)
                          (app/prim p a)))))

(define run-pcd (call load&run))
(define run-adv (lambda (p) (lambda (a)
                      (if (no-run-permission? user)
                          (raise 'no-permission-exception)
                          (app/prim p a)))))

(around read-pcd read-adv
  (around write-pcd write-adv
    (around run-pcd run-adv
      (list read-file write-file run-program))))
```

Since we use static aspects to encapsulate the permissions feature, the aspect will be used when evaluating the bodies of the function, wherever that may occur. Thus, we can safely export these three functions from our library. Second, we would like to add an extra security measure to the *run-program* function. The argument *p* is some client-supplied program, and we may to restrict its access to certain resources. For example, we can ensure that the client program does not open any network connections by using a dynamic aspect. This aspect will dictate that any call to *open-socket* that occurs in its dynamic extent should fail. We add this dynamic aspect in the advice that governs *run-program*:

```
(define no-socket-adv (lambda (p) (lambda (a)
                          (raise 'no-socket-allowed))))

(define run-adv (lambda (p) (lambda (a)
                    (if (no-run-permission? user)
                        (raise 'no-permission-exception)
                        (fluid-around (call open-socket)
                                      no-socket-adv
                          (load&run p))))))
```

This security example illustrates the utility of both static and dynamic aspects. Static aspects allow us to encapsulate cross-cutting features of library functions, and export the functions so that they use the aspect when applied. Dynamic aspects give us control of whatever computations occur within some dynamic extent: in this case, we could catch certain function calls in the extent of an untrusted client's program.

## 3.2 Reusing Aspects

In this section, we study examples of how first-class pointcuts and advice allow greater reuse. First, we examine the difference between pointcut descriptors in our language, and those of AspectJ. In our formulation, pcd's are first-class values: they are predicates over a list of join points. Like all values in a functional language, they can be passed to and returned from functions. In AspectJ, however, the programmer cannot abstract over pcd's; Kiczales et al. explicitly state: "Pointcuts are not higher order, nor are pointcut designators parametric." [11] Are there any advantages to having first-class pointcuts?

Consider a pcd that describes the following join point: any call to the function *func1* where control flowed through functions *f*, *g*, and *h* in that order. This situation might arise when the programmer registers *func1* as a callback function, and she wishes to examine those calls to it that originated from control flow sequence *f*, *g*, *h* in the library. We can write this pcd as follows:

```
(&& (call func1)
    (cflow (&& (call f)
               (cflow (&& (call g)
                          (cflow (call h)))))))
```

Now let's describe the same scenario, but for the function *func2* instead of *func1*. In AspectJ, we would have to write the entire pcd again, due to the lack of parametricity. This duplication of code not only makes the programmer's task more difficult, but likely will induce errors when modifying the code. In our language, on the other hand, we can parameterize the pcd over the function:

```
(define (thru-fgh a-function)
  (&& (call a-function)
      (cflow (&& (call f)
                 (cflow (&& (call g)
                            (cflow (call h))))))))
```

Thus *thru-fgh* consumes a function and returns a pointcut descriptor. We can use *thru-fgh* to create a pcd for both *func1* and *func2*, or indeed for any function:

```
(thru-fgh func1)
(thru-fgh func2)
```

By making pcd's first-class entities in a functional language, we automatically get the greater abstractive capability afforded by parameterization.

We can take this abstraction one level further. In the example above, we used a chain of *cflow*s to represent a path of control flow. We will likely use this control flow pattern beyond just the functions *f*, *g*, and *h*, so we would would like to define a more general pointcut operator: one that takes a list of pcd's and produces a new pcd representing any join point where control flowed successively through each pcd in the list. We'll call this operator *cflow*∗. In our language, we can define *cflow*∗ as a recursive function in terms of *cflow*:

```
(define (cflow∗ lis)
  (if (empty? lis)
      (lambda (jp) true)
      (cflow (&& (first lis)
                 (cflow∗ (rest lis))))))
```

Alternatively, we could define the function using *foldr*:

```
(define (cflow∗ lis)
  (foldr (lambda (this-pcd rest-true)
           (cflow (&& this-pcd rest-true)))
         (lambda (jp) true)
         lis))
```

We can rewrite our above example *thru-fgh* as follows:

```
(define (thru-fgh a-function)
  (&& (call a-function)
      (cflow∗ (list (call f) (call g) (call h)))))
```

The operator *cflow∗* is higher-order pointcut descriptor: it consumes a list of pcd's and produces a new one. Again, the power to define such operator comes for free from defining pcd's as first-class entities in a functional language. We believe this abstractive power represents a significant improvement over the capabilities of AspectJ.

Not only can we define higher-order pointcuts, but we can define higher-order advice. We illustrate one scenario where this ability is useful. Consider the circumstance where we have two aspects: one that logs calls to a function, and one that filters calls to a function based on its argument. The advice for the logging aspect prints out a message before and after the join point:

```
(lambda (p)
  (lambda (a)
    (begin
      (printf "entering fn")
      (app/prim p a)
      (printf "exiting fn"))))
```

The filtering aspect may or may not enter the join point, depending on the value of the argument; its advice is defined as follows:

```
(lambda (p)
  (lambda (a)
    (unless (zero? a)
      (app/prim p a))))
```

What happens if these two aspects both apply to the same join point? There are two possibilities:

1. The filtering advice executes first, and its call to *p* invokes the logging advice. When *a* is zero, it does not call the logging advice (and thus the original function), so nothing is printed.
2. The logging advice executes first, and its call to *p* invokes the filtering advice. When *a* is zero, the logging advice still prints out its messages, even though the pruning advice does not call the original function.

Given these two choices, we probably desire the behavior of the first. How can we ensure this behavior? In AspectJ, the order of aspect weaving depends on the order of their definitions in the source file (though we could use the **dominates** modifier to specify order more precisely). A safer approach would be to combine these two pieces of advice ourselves, so that we have absolute control over their order, and do not have to rely on the implicit ordering of the system. Thus we could write a function that consumes these two advice functions and returns their combination:

```
(define (sequence-advice advice1 advice2)
  (lambda (p)
    (lambda (a)
      ((advice1 (advice2 p)) a))))
```

This function *sequence-advice* is higher-order advice: it consumes two pieces of advice and produces new advice. For more complex aspects, we would need more detailed ways of combining them. In AspectJ, we cannot define new combinators without modifying the internals of aspect weaving. In our language, we have complete control over how to combine multiple aspects that apply to the same join point.

## 4. SEMANTICS

We have developed a formal semantics for pointcuts and advice in a higher-order language, which we will briefly describe here. The important element of these semantics is an *aspect environment*, which maintains a list of active aspects. The constructs **around** and **fluid-around** extend the aspect environment, while function application examines the environment to apply relevant aspects. The remaining parts of the language are standard, except that functions close over both the variable *and* aspect environments.

The aspect environment is a list of triples $\langle V_{pcd}, V_{advice}, scope\rangle$, where $V_{pcd}$ and $V_{advice}$ are (procedure) values, and the $scope$ tag is either `static` or `dynamic`. The expression (**around** $M_{pcd}$ $M_{advice}$ $M_{body}$) evaluates its first two components, adds the resulting values to the aspect environment with a `static` tag, and then evaluates the body in this extended environment. The semantics of **fluid-around** is similar, except that the scope tag is `dynamic`.

Next we turn to function application. Recall that our language has two such constructs: the default one, which weaves aspects into the computation, and a "primitive" application, which does not observe aspects. Although **app/prim** does not invoke aspects, it must create the correct aspect environment in which to evaluate the body of the function. Let $A_{app}$ be the aspect environment present at the call site, and let $A_{fun}$ be the environment extracted from the function's closure. The aspect environment for evaluating the function body then comprises the `dynamic` aspects in $A_{app}$ and the `static` aspects from $A_{fun}$.

Finally we come to the heart of our semantics: how to inject aspects into the computation during function application. This process of "aspect weaving" contains three steps:

1. **Record the join point.** We record that we have entered a new join point by placing a mark on the stack. This mark has no direct effect on the computation; it is simply discarded when the function returns.
2. **Compute the current join points.** We compute the list of current join points, which each pcd takes as an argument. This list can be created by a function over (some concrete representation of) the current continuation.
3. **Check each aspect.** We check each aspect in the aspect environment: if the pcd holds, we apply the advice to the function. After checking all aspects, we apply the resulting function to the original argument. If no pcd evaluates to true, this function application reverts to **app/prim**. For each aspect that holds, the application of the advice to the function also invokes aspect weaving.

We give the formal specification of these semantics as a variation on the CEKS machine [6]; full details will be available in a forthcoming technical report [19].

## 5. IMPLEMENTATION

We demonstrated that we can support several key elements of aspect-oriented programming in a functional language by adding three language constructs. In this section, we will present these constructs as syntactic extensions to the Scheme language. To do this, we will need to employ Scheme's macro system, along with the PLT Scheme facilities for creating new languages. We will also describe continuation marks, and use them to define our language constructs.

## 5.1 Background on PLT Scheme

In Scheme, we can easily define language extensions using its macro system. Scheme macros are effectively functions that rewrite syntax trees; they are more powerful than lexical macros, such as those provided by the C preprocessor, which operate only on strings. *Hygienic* macros ensures that the syntax tree resulting from a transformation does not accidentally capture any variables from the surrounding context [13, 15]. To define macros in PLT Scheme [8], we will use the **syntax-case** form [5], which allows pattern-matching [14] and creates hygienic macros.

Macros themselves are not sufficient for defining our aspect-oriented extensions. As we saw earlier, we must redefine the behavior of function application so that it performs aspect weaving; thus, we are really creating a new language, not merely an extension to Scheme. Fortunately, PLT Scheme's module system provides an easy way to create a new language: the programmer defines a module that exports the syntax definitions for every construct in the language [9]. Our implementation exports the default language constructs from Scheme with a few changes. We define and export the new syntactic forms **around** and **fluid-around**. We also define **app/weave**, the form of function application that weaves aspects, and export it as the default application. We then export Scheme's default function application as **app/prim**.

In order to implement aspect-oriented programming, we need one additional feature of PLT Scheme: continuation marks. Clements, Flatt, and Felleisen introduced continuation marks as a mechanism for implementing an algebraic stepper [3]. The stepper inserts a break point between each evaluation step to show the execution of a program. At each break point, the stepper prints representations of both the current value and the current continuation. Clements et al.'s insight was to mark every computation point with a representation of its action; the stepper can then reconstruct the structure of the continuation by examining these marks at break points.

In terms of language design, these marks require the addition of two primitives. Intuitively, **with-continuation-mark** adds a mark, and **current-continuation-marks** examines the marks. (We will abbreviate these constructs as **w-c-m** and **c-c-m** respectively.) The expression (**w-c-m** *tag* $M_1$ $M_2$) first evaluates $M_1$, then $M_2$, and returns the value of $M_2$. The expression (**c-c-m** *tag*) looks for instances of (**w-c-m** *tag* $V$ ...) in the current continuation, and returns a list of all such $V$'s. For example:

```
(define (fact n)
  (w-c-m 'fact-arg n
    (if (zero? n)
        (begin
          (display (c-c-m 'fact-arg))
          1)
        (* n (fact (sub1 n))))))

(fact 4)
```

prints (0 1 2 3 4).

For implementing a stepper, it was critical that continuation marks preserve tail-call behavior. The semantics of continuation marks dictate that when two consecutive marks are placed with the same *tag* on the stack, the newer one overwrites the older one. Thus, the accumulator equivalent of the factorial implementation above:

```
(define (fact n a)
  (w-c-m 'fact-arg n
    (if (zero? n)
        (begin
          (display (c-c-m 'fact-arg))
          a)
        (fact (sub1 n) (* n a)))))

(fact 4 1)
```

prints a list containing just the number 0, because the continuation mark created at each recursive call overwrites the previous mark. Unfortunately, we do not want this overwriting behavior in our uses of continuation marks. We can ensure that two marks never appear consecutively by inserting an application of the identity function before each **w-c-m** expression. For example, we can transform the accumulator-style definition of *fact* so that no marks disappear:

```
(define (fact n a)
  ( (lambda (x) x)
   (w-c-m 'fact-arg n
     (if (zero? n)
         (begin
           (display (c-c-m 'fact-arg))
           a)
         (fact (sub1 n) (* n a))))))

(fact 4 1)
```

This expression prints the list (0 1 2 3 4) as desired.

Since our uses of continuation marks always want this behavior, our code redefines **w-c-m** to automatically insert an application of (**lambda** (*x*) *x*) as in the above example. All instances of **w-c-m** in the remainder of this paper will use this redefinition.

## 5.2 Implementation of Dynamic Aspects

How can we use continuation marks to define our aspect-oriented extension to Scheme? There is one obvious parallel between aspects and continuation marks: the dynamic nature of join points. Recall that the **cflow** operator allows the programmer to match any join point in the dynamic context. When we enter a new join point, we add a continuation mark containing the data for the join point—in our model, the value of the function. In order to evaluate pcd's, we need the list of all active join points, which we retrieve by examining continuation marks. Both of these events occur during function application. Figure 1 shows the code for **app/weave**. The expression (**w-c-m** 'joinpoint *fun-val* ...) records a join point, and (**c-c-m** 'joinpoint) retrieves the current list of join points.

We also need some way to mimic the aspect environment defined in our semantics. The environment contained both static and dynamic aspects; for now, we will focus on the dynamic aspects. Continuation marks are in fact an implementation of dynamic environments: **w-c-m** extends the dynamic environment with a new value, and **c-c-m** returns all values. When we encounter a dynamic aspect, we add it to the dynamic environment with the expression (**w-c-m** 'dynamic-aspect *aspect* ...). When we need to weave aspects during function application, we retrieve the list of all dynamic aspects via (**c-c-m** 'dynamic-aspect). The definitions of **fluid-around** and **app/weave** exhibit this use of continuation marks.

We now have the two pieces of information we need to weave dynamic aspects: the list of current join points and the active dynamic aspects. At function application, we iterate over each aspect. If the aspect's pointcut descriptor returns true when applied to the join point list, we apply the aspect's advice to the function. The definition of *weave* demonstrates the details of this algorithm.

```
(module aspect-scheme mzscheme
  (define-struct aspect-pair (pcd advice))

  (define-syntax (app/weave stx)
    (syntax-case stx ()
      [(_ f a ...) (syntax (app/weave/rt f a ...))]))

  (define (app/weave/rt fun-val . arg-vals)
    (if (primitive? fun-val)
      (apply fun-val arg-vals)
      (w-c-m 'joinpoint fun-val
        (let* ([jp (c-c-m 'joinpoint)]
               [aspects (current-aspects)])
          (apply (weave fun-val jp aspects)
                 arg-vals)))))

  (define (current-aspects)
    (c-c-m 'dynamic-aspect))

  (define (weave fun-val jp aspects)
    (if (empty? aspects)
      fun-val
      (let ([r (weave fun-val jp (rest aspects))]
            [a (first aspects)])
        (if ((aspect-pair-pcd a) jp)
          (lambda vs (apply app/weave/rt
                            ((aspect-pair-advice a) r)
                            vs))
          r))))

  (define-syntax (fluid-around stx)
    (syntax-case stx ()
      [(_ pcd advice body)
       (syntax (w-c-m 'dynamic-aspect
                      (make-aspect-pair pcd advice)
                 body))]))

  (provide (all-from-except mzscheme #%app)
           (rename app/weave #%app)
           (rename #%app app/prim)
           fluid-around))
```

**Figure 1: Dynamic aspects**

## 5.3 Implementation of Static Aspects

Although continuation marks map well to two features of aspect-oriented programming—join points and dynamic aspects—they do not obviously help in implementing static aspects. Consider these two examples; the latter one was our example in section 2.2 which demonstrated statically-scoped aspects:

```
((fluid-around (call add2) trace-advice (lambda (v) (add2 v))) 7)

((around (call add2) trace-advice (lambda (v) (add2 v))) 7)
```

In the first example, the dynamic aspect is not in scope when *add2* is applied to 7. Our macro produces this behavior: the continuation mark corresponding to the **fluid-around** disappears when the body (**lambda** (*v*) (*add2 v*)) returns, before the application of *add2*. The second example, however, declares a static aspect, which *is* in scope whenever the body of (**lambda** (*v*) (*add2 v*)) executes. The **around** expression also stores the aspect in a continuation mark, but that mark will disappear when the body of the **around** returns; we will lose the static aspect.

In order to achieve the correct semantics for **around**, we need to transform each **lambda** expression in the program so that it closes over the aspects at its definition site, and reinstates these aspects during the execution of its body. Consider a single-argument function (**lambda** (*x*) *body*); we wish to transform this to an equivalent function that stores the static aspects:

```
(lambda (x) (w-c-m 'static-aspect ... body))
```

What aspects belong in the elided expression? We want all static aspects that were active at the site of the function's definition. We explain this through an example. Consider the program

```
(let ([f (around pcd₁ advice₁
           (lambda (x)
             (around pcd₂ advice₂
               (g x))))])
  (around pcd₃ advice₃
    (f 0)))
```

The procedure bound to *f* needs to close over the aspect with the pcd $pcd_1$. The transformed procedure captures this as follows:

```
(lambda (x)
  (w-c-m 'static-aspect (make-static-env
                          (list
                            (make-aspect-pair pcd₁ advice₁)))
    (around pcd₂ advice₂
      (g x))))
```

At the time of invoking *f*, the stack currently contains just one mark for an aspect, that for $pcd_3$. Invoking *f* pushes the static environment onto the stack, so it now has two marks. The original body of *f* pushes one more mark, that for $pcd_2$. Because each **around** generates a continuation mark for the aspect it declares, invoking (**c-c-m** 'static-aspect) at the point of applying *g* returns

```
(list (make-aspect-pair pcd₂ advice₂)
      (make-static-env (list (make-aspect-pair pcd₁ advice₁)))
      (make-aspect-pair pcd₃ advice₃))
```

Of these, all aspects higher on the stack than the highest static environment were defined in scopes that extend that of an invoked procedure (in this case, just $pcd_2$). In contrast, all those lower on the stack than the highest static environment can no longer be in scope (by definition of static scoping), and we must therefore ignore them—in this case, that of $pcd_3$. Therefore, the **around** macro employs a procedure that elides all aspects beyond the highest static aspect:

```
(define (active-static-aspects lis)
  (if (empty? lis)
    '()
    (if (static-env? (first lis))
      (static-env-lis (first lis))
      (cons (first lis) (active-static-aspects (rest lis))))))
```

Figure 2 defines the macro **lambda/static**, which implements the transformation described above. We export **lambda/static** as the default **lambda** for our language. Notice that we also update the definition of *current-aspects*, so that it considers static aspects in addition to dynamic ones.

```
(define (current-aspects)
  (append (c-c-m 'dynamic-aspect)
          (active-static-aspects (c-c-m 'static-aspect))))

(define-syntax (around stx)
  (syntax-case stx ()
    [(_ pcd advice body)
     (syntax (w-c-m 'static-aspect
                    (make-aspect-pair pcd advice)
               body))]))

(define-syntax (lambda/static stx)
  (syntax-case stx ()
    [(_ params body ...)
     (syntax
      (let ([env (active-static-aspects (c-c-m 'static-aspect))])
        (lambda params
          (w-c-m 'static-aspect (make-static-env env)
            (begin body ...)))))]))
```

**Figure 2: Static aspects**

The definitions given in Figures 1 and 2 are now executable code—they correctly interpret the examples given in this paper.

## 6. RELATED WORK

While the earlier work on aspects [12] was defined for languages like Common Lisp that do offer higher-order programming facilities, the aspects themselves were defined broadly through generalized weavers. This work did not explicitly distinguish between different scoping mechanisms for aspects. While it is perhaps possible to define these scopes using particular weavers, the work does not identify this concern or discuss its potential.

AspectJ [11] is the de facto standard for aspect-oriented programming. It defines a rich set of join points for describing points in the execution of a program. Since Java is a statically-typed language, AspectJ also requires and enforces type declarations when defining aspects. The programmer can also use types in pointcut descriptors, which is extremely useful in conjunction with wildcards. AspectJ's support for software development includes a compiler that produces standard Java bytecode, and extensions to programming environments that enable the programmer to browse aspect hierarchies.

Wand, Kiczales, and Dutchyn [20] present a denotational semantics for aspect-oriented programming. Like us, they study an aspect-oriented extension to an untyped language; however, they only support first-order procedures. Although we have developed an operational semantics that includes higher-order functions, many of our ideas derive from their work, such as the use of an aspect environment, and the characterization of advice as procedure transformers.

Bauer, Ligatti, and Walker [1] present a model for language-based security, where an outside program monitors the execution of an untrusted program. Their security policies have the same structure as aspects: they comprise a set of actions to intercept in a program's execution, and a policy that can modify the computation of these actions. Furthermore, the security policies are first-class values, and they give examples of parametric and higher-order policies. Their system is similar to aspect-oriented programming, except that they do not support the same range of pointcuts; notably, they do not provide a means of examining control flow.

## 7. CONCLUSION

As aspect-oriented software design grows in popularity, more languages will need to support this style of development. Recent work on defining a semantics for pointcuts and advice [20] is especially valuable, because it explicates the essence of these kinds of aspects, making it easier to port this style of programming between languages. Because the semantics is defined for first-order languages, however, it fails to document how to define AspectJ-like features for languages with first-class and higher-order procedures. As the family of languages with these features includes not only academic languages such as Scheme and ML but also industrially popular languages such as Python and Perl, defining aspects in this context takes on immediacy and importance.

Higher-order languages present both challenges and benefits for aspects. On the one hand, they force the designer to carefully address issues of scope that do not arise in first-order languages. Not only do procedural entities no longer necessarily have names, programmers can now distinguish between their loci of definition and of use. One the other hand, these distinctions of scope make it possible to define a much richer variety of policies than is possible in a first-order aspect language. In particular, programmers can now define both dynamic aspects traditional to AOP and static aspects that can enforce policies defined within modules, e.g., capture common security-control paradigms.

In this paper, we present a description of aspects for higher-order languages. We mimic the operators of AspectJ but implement them in the context of the Scheme programming language. We also describe the implementation of this language. The implementation exploits two novel features of our Scheme system—continuation marks and language-defining macros—that do not interfere, and indeed integrate well, with traditional tasks such as separate compilation and the use of the DrScheme development environment [8]. This makes it very convenient for programmers to exploit aspects to improve program designs without changing their program development methodology. In addition, continuation marks are implemented efficiently, so programmers are not penalized for their use.

There are many directions for future work. While we have explained how aspects should behave in higher-order languages, we have not provided an account of pointcuts and advice in languages with even richer (and increasingly popular) control primitives such as continuations. We have also deliberately neglected type system questions, particularly the kinds of parametric polymorphism that aspects induce, and other forms of static validation. Finally, we have paid relatively little attention to the run-time cost of using aspects and should seek ways to optimize them (perhaps by shifting some work to compile-time) to make them minimally intrusive.

### *Acknowledgements*

Thanks to Matthias Felleisen and John Clements for discussions on this research.

## 8. REFERENCES

[1] Lujo Bauer, Jarred Ligatti, and David Walker. A calculus for composing security policies. Technical Report TR-655-02, Princeton University, 2002.

[2] Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001.

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

[4] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[5] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

[6] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. http://www.cs.utah.edu/plt/publications/pllc.pdf.

[7] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA*, October 2000.

[8] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

[9] Matthew Flatt. Composable and compilable macros: You want it when? In *ACM SIGPLAN International Conference on Functional Programming*, 2002.

[10] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck. Handling crosscutting constraints in domain-specific modeling. *Communications of the ACM*, 44(10):87–93, 2001.

[11] 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.

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, June 1997.

[13] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. In *ACM Symposium on Lisp and Functional Programming*, pages 151–161, 1986.

[14] Eugene E. Kohlbecker and Mitchell Wand. Macros-by-example: Deriving syntactic transformations from their specifications. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 77–84, 1987.

[15] Eugene E. Kohlbecker Jr. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, August 1986.

[16] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10):39–41, 2001.

[17] Paniti Netinant, Tzilla Elrad, and Mohamed E. Fayad. A layered approach to building open aspect-oriented systems: a framework for the design of on-demand system demodularization. *Communications of the ACM*, 44(10):83–85, 2001.

[18] Greg Sullivan. Aspect-oriented programming with reflection and meta-object protocols. *Communications of the ACM*, 44(10):95–97, 2001.

[19] David B. Tucker and Shriram Krishnamurthi. A semantics for pointcuts and advice in higher-order languages. Technical Report CS-02-13, Department of Computer Science, Brown University, December 2002.

[20] 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.