Semantics and Scoping of Aspects in Higher-Order Languages

Christopher Dutchyn[a,*], David B. Tucker[b], Shriram Krishnamurthi[b,1]

[a]*Department of Computer Science, University of British Columbia*

[b]*Computer Science Department, Brown University*

Abstract

Aspect-oriented software design will need to support languages with first-class and higher-order procedures, such as Ruby, 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 provide an operational semantics, based on an extended CEKS machine, that gives a formal account of dynamic and static aspects. We implement the 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.

Key words: Scheme, aspect, join point, pointcut, advice, higher-order, CEKS, abstract machine

An earlier version of this work appeared at the ACM SIGPLAN Conference on Aspect-Oriented Software Development, 2003 [35].

* Email: `cdutchyn@cs.ubc.ca`

1 This research was partially supported by NSF grants ESI-0010064, ITR-0218973, and CCF-0447509.

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 prevent modular assembly: the programmer cannot simply add or delete these features from a program, since they are not separable units.

Recently, many researchers have proposed aspect-oriented software development as a method for organizing cross-cutting features [2, 6, 20, 24, 29, 31, 34]. In particular, Kiczales et al. [24] have presented aspect-oriented programming (AOP); in this paradigm, the fragments of any given feature precipitate into a separate component, called an *aspect*. In addition to containing the code necessary for a feature, the aspect must indicate how this code should combine with other modules to provide the desired behavior. Kiczales et al. also implemented a practical aspect-oriented extension to Java, called AspectJ, which allows the programmer to define aspects and integrates them into a program [23].

Most 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 issues motivate our investigation of this topic. On the 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 Ruby, 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 in 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. This foundation might enable us to apply parametricity to define more general aspects, and develop aspect combinators by employing higher-order functions. The interaction between AOP and functional programming therefore merits careful investigation.

The two main challenges to adding AspectJ-style aspects to a functional language are the specification of aspects and the definition of the scope of an aspect's applicability. First, we need to decide how to specify aspects. Are they new kinds of values? Do we need a sub-language for describing where aspects apply? Second, we must address the issue of scope. Unlike first-order languages, where all procedures and aspects are declared at the top level and have very broad scope, most higher-order languages permit definitions to be introduced at any point and have more limited 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. In section 4, we describe in detail AspectScheme, a lightweight implementation of aspects for Scheme, comment on enhancements that are available, and examine potential efficiency concerns. Section 5 informally presents the semantics for our extension and connects the lightweight implementation to this semantic description. Section 6 discusses related work, and section 7 concludes.

2 Defining Pointcuts and Advice

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 Definitions in 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, including some points that the programmer perhaps did not anticipate in advance.

These execution points are called *join points*, and they depend both on the underlying language and the aspect-oriented extension to it. AspectJ defines a set of possible join points for Java programs, 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.

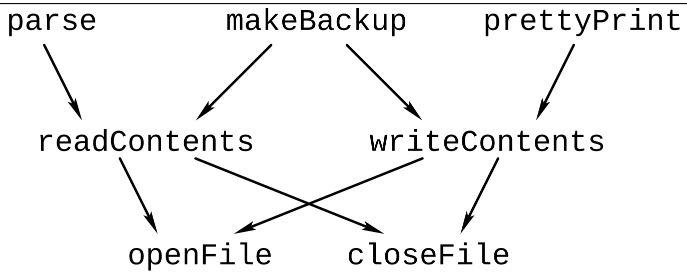

Fig. 1. Call graph for a edit-buffer library

Each join point presents a locus for a feature to affect a computation. The effect might be as simple as writing some trace message to output or might be as detailed as entirely supplanting the computation about to occur. The specification of an aspect therefore has two attributes: the *pointcut* 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 edit-buffer manipulation library with the call graph shown in Figure 1.

The following AspectJ pointcut refers to any join point where the method *closeFile* is about to be called from the body of method *writeContents*:

call(void *closeFile*()) *&&* **withincode**(void *writeContents*())

AspectJ provides many means for defining pointcuts. Simple pointcuts, such as those matching method calls and variable accesses, can combine via boolean connectives to create complex pointcuts, 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 pointcut describes all calls to *closeFile* occurring as part of a *makeBackup* call:

call(void *closeFile*()) *&&* **cflow(call**(void *makeBackup*()))

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

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

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.[2] **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 *readContents*, prints the returned value, and returns the value to the original context:

```
bool around() : call(bool readContents()) {
   bool b = proceed();
   System.out.println("readContents returned " + b);
   return b;
}
```

An *aspect* combines pointcuts, a characterization of join points of concern, and advice, describing the modified behavior at those points. The pointcut and advice are not strictly independent entities in AspectJ. The pointcut may pattern match against values in the join points it specifies; the advice can then refer to these values in its code. Aspect definitions may use this facility to capture the arguments to a method call; for example, we can ensure that the application never opens a *.java* file.

```
bool around(String s) : call(bool openFile(String)) && args(s) {
   if (s.endsWith(".java"))
      return false;
   else
      return proceed(s);
}
```

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

- AspectJ defines a special language for pointcuts; 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.
- Aspects comprise multiple pointcut and advice pairs, with the scope of application encompassing the entire program.

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

[2] AspectJ supports three **after** advice patterns: **after returning** executes advice only after a normal return; **after throwing** executes advice only when an exception occurs; and **after** executes advice after the join point regardless of normal or exceptional return.

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 [17, 19] as the sandbox for our experimentation because its macro system provides especially powerful support for linguistic extensions. We will rely on these for developing a lightweight implementation of aspects in section 4.

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? Do we make aspects first-class values? Second, since functions may be nameless, how do we identify them? Third, what scope does an aspect have; that is, when should a given aspect be in force?

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 recognize procedure applications as join points. But, each join point is either top-level (with empty context) or nested within another active procedure application. Therefore, we represent a join point in context as a list of procedures: the about-to-be-applied procedure is at the head, and its context (another join point in context) as the tail. Then, we define a pointcut as a predicate over a join point in context, and advice as a join point (procedure) transformer.

Consider the pointcut we saw earlier, but in the context of a functional language: the set of join points representing calls to *close-file* from *write-contents*. In our model, a pointcut consumes a join point in context—a list of procedures 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 pointcut as follows:

```
(λ (jp∗)
  (and (eq? close-file (first jp∗))
       (not (empty? (rest jp∗)))
       (eq? write-contents (second jp∗))))
```

Our primitive *eq?* deems two functions equal if they have the same source location and close over identical environments.[3] We use *eq?* to solve the problem of identifying functions in pointcuts; in the above code, for example, both the variable *close-file* and the expression (*first jp∗*) evaluate to functions, which can then be compared using *eq?*. We will discuss the semantics in detail in section 5.

[3] We compare environments "by reference"; for more details see section 5.3.

```
(define ((call f) jp*)
  (eq? f (first jp*)))

(define ((within f) jp*)
  (and (not (empty? (rest jp*)))
       (eq? f (first (rest jp*)))))

(define ((cflow pc) jp*)
  (and (not (empty? jp*))
       (or (app/prim pc jp*)
           (app/prim (app/prim cflow pc) (rest jp*)))))

(define ((cflowbelow pc) jp*)
  (and (not (empty? jp*))
       (app/prim (app/prim cflow pc) (rest jp*))))

(define ((&& pc1 pc2) jp*)
  (and (app/prim pc1 jp*)
       (app/prim pc2 jp*)))

(define ((|| pc1 pc2) jp*)
  (or (app/prim pc1 jp*)
      (app/prim pc2 jp*)))

(define ((! pc) jp*)
  (not (app/prim pc jp*)))
```

Fig. 2. Standard pointcut operators[4]

The other pointcut we defined earlier denoted all calls to *close-file* that originated from *make-backup*. In our language, we define this pointcut as follows:

```
(λ (jp*)
  (and (eq? close-file (first jp*))
       (app/prim memq make-backup (rest jp*))))
```

where *memq* checks, with *eq?*, whether an element is a member of a list. The syntactic form **app/prim** performs a “primitive application”; that is, it

[4] For brevity, we adopt MIT Scheme’s curried style of procedure definition [21]. The equivalent R^{5}RS Scheme [22] definition for *call* is

```
(define call
  (λ (f)
    (λ (jp*)
      (eq? f (first jp*)))))
```

applies a function to an argument without examining whether aspects apply. Informally, the pointcut says "return true if *close-file* is the first join point and *make-backup* occurs in the rest of the join point list."

Since pointcuts are first-class values, we can define standard pointcut operators without any special language support, as shown in Figure 2. That figure also shows the importance of **app/prim**. If we had defined *cflow* with (*pc jp*∗) in place of the boxed expression in Figure 2, that application would itself invoke aspect weaving, which in turn would evaluate the same *cflow* pointcut, leading to an infinite loop.

Using these standard pointcuts, we can rewrite the two previous examples as follows: [5]

```
(&& (call close-file) (within write-contents))

(&& (call close-file) (cflow (call make-backup)))
```

Notice that these definitions closely resemble the original AspectJ code.

We now turn to the definition of advice. We will focus on **around** advice, since it is strictly more general than both **before** and **after**. [6]

We define advice as a procedure transformer: it consumes a procedure (the current join point) and returns a new procedure to use in its place. This formulation of advice is similar to the denotational semantics of advice given by Wand et al. [37] for a first-order procedural language. [7]

For example, the following advice prints a message before invoking the original function *jp* on the original argument *a*:

```
(λ (jp)
  (λ (a)
    (printf "Calling open-file")
    (app/prim jp a)))
```

[5] Also for brevity, in the rest of the paper we omit **app/prim** when applying the standard pointcut operators given in Figure 2.
[6] In consideration of higher-order languages that support exceptions, we note that the three variants of **after** are simple advice arrangements within an exception-handling expression comprising the body of the **around** advice.
[7] We chose not to provide the join point context to advice because it is immutable for our purposes; any arguments and values have already affected that context. An enhanced version of the language, discussed in section 4.4, supplies read-only access to arguments of join points matched by a pointcut (just like **target** and **args** in AspectJ).

In this case, the use of **app/prim** ensures that applying *jp* 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 (jp)
  (λ (a)
    (let ([s (app/prim jp a)])
      (printf "value is ~a" s)
      s)))
```

The first parameter, *jp*, 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 the value, and returns it.

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

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

This advice captures the original procedure, *jp*, and provides a replacement examines the argument *a*, and decides whether to simply return the default value, `""`, or to proceed. Here, we again employ **app/prim** to capture the behavior of AspectJ's **proceed**, which applies the original function without performing any additional aspect weaving at the join point. 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 procedure definitions in its pointcuts, 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 to our language a new expression for "around" aspects:

```
(around pc advice
  body)
```

Informally, this expression means "the aspect defined by *pc* and *advice* applies in *body*". For example, we could write:

```
(let ([open-file (λ (f) . . . )]
      [trace-advice (λ (jp)
                      (λ (a)
                        (printf "Calling open-file")
                        (app/prim jp a)))])
  (around (call open-file) trace-advice
    (list (open-file "vancouver")
          (open-file "whistler"))))
```

While the **around** construct may ostensibly seem straightforward, we have 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 regardless of where it is used. If the body of the declaration is a procedure, then the aspect applies in every use of that 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.

Concretely, consider the following use of the above *trace-advice*:

```
(around (call open-file) trace-advice
  (open-file "vancouver"))
```

In this case, the advice is applied statically. Therefore, the aspect is in force when the application of *open-file* to "vancouver" occurs in the text of the body. As a result, the advice prints a message before executing *open-file*. However, not all applications that occur while evaluating the body appear in the text of the body. Consider the following:

```
(let ([static-traced-open (around (call open-file) trace-advice
                                   (λ (f) (open-file f)))])
  (static-traced-open "vancouver"))
```

The evaluation of this expression also prints a trace message, because the application of *open-file* to *f* 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-vancouver (λ (f) (f "vancouver"))])
  (around (call open-file) trace-advice
    (apply-to-vancouver open-file)))
```

Since we may wish to define aspects that *do* apply in the above example, we also allow *dynamically* scoped aspects. The expression (**fluid-around** *pc 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-vancouver (λ (f) (f "vancouver"))])
  (fluid-around (call open-file) trace-advice
    (apply-to-vancouver open-file)))
```

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

Next, 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:

```
(let ([dynamic-traced-open (fluid-around (call open-file) trace-advice
                                         (λ (f) (open-file f)))])
  (dynamic-traced-open "vancouver"))
```

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

To clarify, we consider one final example:

```
(let ([traced-to-vancouver (around (call open-file) trace-advice
                                    (λ (f) (f "vancouver")))])
  (traced-to-vancouver open-file))
```

In this case, the tracing message is emitted. Careful examination shows that all calls to the procedure bound to *open-file* (also bound to f) occur lexically within the advice body.

Our aspects combine pointcuts, advice, and new **around** and **fluid-around** expressions to provide an AOP language similar to AspectJ. In particular, we retain two key properties of AOP, quantification and obliviousness [15, 16]. The first property, *quantification*, describes the range of entities that may be affected by advice, and is embodied in our definition of a pointcut and the clearly-delimited scoping provided by our **around** and **fluid-around** expressions. The second property, *obliviousness*, allows a programmer to alter program behavior without changing the original code and without anticipating any future changes. For example, with the following application

```
(define (backup-system)
  (for-each make-backup
            (list "vancouver" "whistler" "lillooet")))
```

we can ensure that tracing occurs by defining the following advice:

```
(fluid-around (&& (call close-file) (cflow (call make-backup)))
              trace-advice
  (backup-system))
```

That is, we are able to modify the behavior of a program without leaving any hooks in it.

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.

```
(module os-api ;; names the module and
          mzscheme ;; specifies its implementation language
  (define (run-program p)
    (if (no-run-permission? user)
        (raise 'no-permission-exception)
        (load&run p)))

  (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)])
          ...)))

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

Fig. 3. An Operating System API

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 operations: *run-program*, *read-file* and *write-file*. The original code is organized by operation, with security checks scattered throughout, as seen in Figure 3.

First, we would like to separate core functionality from permissions checking. The core functionality is:

```
(module os-api aspect-scheme
  (define (run-program p)
    (load&run p))

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

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

We recognize the situations where permissions need to be checked: running a program within *run-program* and opening a file within *read-file* or *write-file*. The join points matching these situations are easily written as the following pointcuts:

```
(define run-pc (&& (call load&run) (within run-program)))
(define read-pc (&& (call open-file) (within read-file)))
(define write-pc (&& (call open-file) (within write-file)))
```

Permissions checking is clear: whenever a join point is matched, apply the corresponding permissions check and then either raise an exception or allow the API call to proceed. We write a general checking advice procedure, and specialize it for each permissions check:

```
(define (((make-perm-check-adv perm-check?) jp) a)
  (if (perm-check? user)
      (raise 'no-permission-exception)
      (app/prim jp a)))

(define run-adv (make-perm-check-adv no-run-permission?))
(define read-adv (make-perm-check-adv no-read-permission?))
(define write-adv (make-perm-check-adv no-write-permission?))
```

Finally, we apply the aspects to the API procedures and export the resulting protected procedures:

```
(define secure-run (around run-pc run-adv (λ (p) (run-program p))))
(define secure-read (around read-pc read-adv (λ (f) (read-file f))))
(define secure-write (around write-pc write-adv (λ (f) (write-file f))))

(provide (rename secure-run run-program)
         (rename secure-read read-file)
         (rename secure-write write-file)))
```

We have clearly separated the security concerns from the original API, enabling the developer to focus on each part separately. Since we use static aspects to encapsulate the permissions feature, the corresponding permission-checking aspect will be applied when evaluating the bodies of the function, wherever that may occur. Thus, we can safely export these three functions from our API module.

Second, we would like to add an extra security measure to the *run-program* function. Suppose the argument *p* is some client-supplied program, e.g. a servlet, and we wish to prohibit its access to certain resources, say opening a network socket. Ordinarily, this would entail providing a distinct *open-socket* routine in our API, or altering that routine's code to recognize illegal uses. But, we can employ a dynamic aspect to ensure that the client program does not open any network connections. This aspect will dictate that any call to *open-socket* that occurs in its dynamic extent of a call to *load&run* within *run-program* should fail by raising an exception. The following code implements this behavior by adding another advice to the *run-pc* pointcut that installs the dynamic aspect that traps calls to *open-socket*.

```
(define open-socket-pc (call open-socket))

(define ((no-socket-adv jp) a) (raise 'no-socket-allowed))

(define ((restrict-run-adv jp) p)
  (fluid-around open-socket-pc no-socket-adv
    (app/prim jp p)))

(define more-secure-run (around run-pc restrict-run-adv
                                 (λ (p) (secure-run p))))

(provide (rename more-secure-run run-program)
           ...))
```

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 trap certain function calls in the execution of an untrusted client's program. This leads to some initial straightforward design rules for determining aspect scope:

- Statically scoped aspects are appropriate for altering the behavior of join points which are lexically visible within the advice body, including those frozen within unapplied lambdas which escape from the advice body.
- Dynamic aspect scoping is applicable when the join points of interest can only be characterized within the dynamic control flow of the advice body (for example, the source is unavailable, or is lexically external) or if the modified behavior should not be permanently engrained in the unapplied procedures.

Overall, aspect scope decisions are largely determined by the existing program modularity: are the join points of interest a static part of the advice body or dynamically present in the advice body? Our aspect that restricts resource access required both kinds of scoping. A dynamic aspect was required to ensure all *open-socket* calls in the control flow of *load&run* were trapped, including those found in *load&run* and p. A static aspect was employed to identify the calls of *load&run* that needed the dynamic aspect to enforce resource restrictions.

3.2 An Algebraic View of Aspects

In this section, we study examples of how first-class pointcuts and advice allow greater reuse. First, we examine the difference between pointcuts in our language, and those of AspectJ. In our formulation, pointcuts 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 pointcuts; Kiczales et al. explicitly state: "Pointcuts are not higher order, nor are pointcut designators parametric" [23]. Are there any advantages to having first-class pointcuts?

Consider a pointcut that describes the following join point: any call to the function $func_1$ where control flowed through functions f, g, and h in that order. This situation might arise when the programmer registers $func_1$ 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 pointcut as follows:

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

Now let's describe the same scenario, but for the function $func_2$ instead of $func_1$. In AspectJ, we could avoid error-prone code duplication by declaring an abstract aspect and a named pointcut to get an extensible pointcut. But, this requires either up-front planning, or re-implementation of the original aspect. In contrast to the lightweight refactoring available to higher-order constructs, this is a heavyweight solution that can bring up cumbersome inheritance issues. As expected with higher-order languages, we can parameterize the pointcut 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. We can use *thru-fgh* to create a pointcut for both $func_1$ and $func_2$, or indeed for any function:

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

By making pointcuts 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 pointcuts and produces a new pointcut representing any join point where control flowed successively through each pointcut in the list. We'll call this operator *cflow∗*. In our language, we can define *cflow∗* as a recursive function in terms of *cflow*:[8]

```
(define (cflow* lis)
  (if (empty? lis)
    (λ (jp*) true)
    (cflow (&& (first lis) (app/prim cflow* (rest lis))))))
```

We can rewrite our above example, *thru-fgh*, using *cflow∗* as follows:

```
(define (thru-fgh a-function)
  (&& (call a-function)
      (app/prim cflow* (app/prim list (call f) (call g) (call h)))))
```

The operator *cflow∗* is a higher-order pointcut: it consumes a list of pointcuts and produces a new one. Our higher-order pointcuts allow us to, for instance, design sophisticated security mechanisms that are difficult for AspectJ to express. For example, we can prevent *open-file* from executing except within an unbroken sequence of trusted procedures ending with a privileged procedure.

[8] Recall that we omit **app/prim** when applying the standard pointcuts given in Figure 2.

```
(define protected-open-file
  (around (&& (call open-file)
              (not (app/prim until trusted? privileged?)))
          (raise 'privilege-exception)
          (λ (f) (open-file f))))
```

This definition depends on the recursively-defined pointcut *until*:

```
(define (((until pc_1 pc_2) jp*)
  (and (not (empty? jp*))
       (or (app/prim pc_2 jp*)
           (and (pc_1 jp*)
                (app/prim (app/prim until pc_1 pc_2)
                          (rest jp*)))))))
```

Again, the power to define such operators comes for free from defining pointcuts 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:

```
(define ((logging-adv jp) a)
  (printf "entering fn")
  (let ([v (app/prim jp a)])
    (printf "exiting fn")
    v))
```

The filtering aspect may or may not enter the join point, depending on the value of the argument; its advice generator, abstracted over the predicate *p?*, is defined as follows:

```
(define (((make-filtering-adv p?) jp) a)
  (if (not (p? a))
      (app/prim jp a)))

(define filter-zero-adv (make-filtering-adv zero?))
```

What happens if *logging-adv* and *filter-zero-adv* advice both apply to the same join point? There are two possibilities:

(1) The filter-zero advice executes first, and its **app/prim** of *jp* 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 **app/prim** of *jp* invokes the filter-zero 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 **declare precedence** construct 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 are able write a function that consumes these two advice functions and returns their combination:

```
(define (((sequence-advice adv1 adv2) jp) a)
  ((app/prim adv1 (app/prim adv2 jp)) 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.

In summary, extending aspects to higher-order languages provides a number of benefits. With higher-order pointcuts, we can abstract and parameterize the description of join points of interest more easily. With higher-order advice, we can provide greater reuse of and finer control over behavioral changes introduced by the aspects. Now we turn our attention to how to implement these facilities.

4 Implementation

We demonstrated that we can support several key elements of aspect-oriented programming in a functional language by adding three language constructs—**around**, **fluid-around**, and **app/prim**. In this section, we will present these constructs as syntactic extensions to the Scheme language by employing the Scheme macro system along with the PLT Scheme facilities for creating new languages. We will also describe the continuation marks facility we use to define these constructs, and present its role in their definition.

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 ensure that the syntax tree resulting from a transformation does not accidentally capture any variables from the surrounding context [25, 26]. We will use the **syntax-case** form [13] in PLT Scheme, which allows pattern-matching [27] 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 [18]. 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 capabilities, 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 [5]. 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.

The mechanism of continuation marks introduces two new language primitives. Intuitively, **with-continuation-mark** (**w-c-m**) adds a mark, and **current-continuation-marks** (**c-c-m**) examines the marks. 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) because the non-tail calls to *fact* extend the continuation to ensure the multiplication happens. These extensions provide separate places to apply the mark for each n, so all marks are found.

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

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

```
(facta 4 1)
```

prints a list containing just the number (0). To see this, recall that tail calls do not extend the continuation. That means the same continuation is always present at the **w-c-m** and the continuation mark applied 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 *facta* so that no marks disappear:

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

(facta 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. This has the effect of extending the continuation with another frame which simply returns the value it awaits. All instances of **w-c-m** in the remainder of this paper will assume this redefinition.

4.2 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 pointcuts, 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 4 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*

```
(module aspect-scheme mzscheme
  (require (only (lib "list.ss") foldr first rest empty?))

  (define-struct aspect (pc adv))

  (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
          (apply (weave fun-val (c-c-m 'joinpoint) (current-aspects))
                 arg-vals))))

  (define (weave fun-val jp* aspects)
    (foldr (λ (a r)
             (if ((aspect-pc a) jp*)
                 ((aspect-adv a) r)
                 r))
           fun-val
           aspects))

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

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

  (define current-aspects current-dynamic-aspects)

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

Fig. 4. Extending Scheme with dynamically-scoped aspects

demonstrate 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 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.

4.3 Implementation of Static Aspects

Although continuation marks help in implementing dynamic aspects, they do not obviously help in implementing static aspects. Recall two examples from section 2.2 which distinguish dynamically- and statically-scoped aspects:

```
(let ([dynamic-traced-open (fluid-around (call open-file) trace-advice
                                         (λ (f) (open-file f)))])
  (dynamic-traced-open "vancouver"))
```

```
(let ([static-traced-open (around (call open-file) trace-advice
                                  (λ (f) (open-file f)))])
  (static-traced-open "vancouver"))
```

In the first example, the dynamic aspect is not in scope when *open-file* is applied to "vancouver". Our macro ensures this behavior: the continuation mark corresponding to the **fluid-around** disappears upon evaluation of (λ (*f*) (*open-file f*)), before the application of *open-file*. But, the second example declares a static aspect which *is* in scope for the body of (λ (*f*) (*open-file f*)).

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. We choose to write the current static aspects into the value of the continuation mark keyed by 'static-aspects, and provide *current-static-aspects* to access them. Then, a lambda expression resembling:

```
(λ (x) . . . )
```

is transformed into

```
(let ([aspects (current-static-aspects)])
  (λ (x)
    (w-c-m 'static-aspects aspects
      . . . )))
```

We provide this transformation of **lambda** as our **lambda/static** macro.

As we are storing the entire list of static aspects in the most-recent 'static-aspects continuation mark, we need *current-static-aspects* to return only that most-recent mark's content. Note that our definition of *current-static-aspects* does this.

```
(define (current-static-aspects)
  (let ([aspectss (c-c-m 'static-aspects)])
    (if (empty? aspectss)
        '()
        (first aspectss))))
```

Now we turn to the problem of constructing the list of current static aspects that corresponds to lexical scoping. Consider the following program structure, involving nested static aspects:

```
(around pc_1 adv_1
  ...; section 1
  (around pc_2 adv_2
    ...); section 2
  ...); section 3
```

Aspect ⟨Aspect pc_1, adv_2⟩ should be available for any closures created in all three elliptic sections, but ⟨Aspect pc_2, adv_2⟩ applies only in section two. Therefore, we define **around** as a macro that obtains the current static aspects and extends them by adding a new continuation mark keyed by 'static-aspects. The transformed example becomes:

```
(w-c-m 'static-aspects
          (cons (make-aspect pc_1 adv_1)
                (current-static-aspects))
  ...; section one
  (w-c-m 'static-aspects
            (cons (make-aspect pc_2 adv_2)
                  (current-static-aspects))
    ...); section two
  ...); section three
```

Our **around** macro performs this transformation. It is instructive to note that for section three of the example, the continuation mark labeled 'static-aspects will have reverted back to the original mark containing only ⟨Aspect pc_1, adv_1⟩. In this way, the lexical scoping of the program is maintained in the static aspects.

Figure 5 contains our implementation of static aspects. It defines the macros **lambda/static** and **around** which implement the transformations described above. Notice that we also update the definition of *current-aspects*, so that it considers both static and dynamic aspects. It also exports **lambda/static** as the default **lambda**.

```
(module aspect-scheme mzscheme
  ;; previous dynamic aspects part elided

  ;; statically-scoped aspects
  (define-syntax (around stx)
    (syntax-case stx ()
      [(_ pc adv body)
       (syntax
        (w-c-m 'static-aspects
               (cons (make-aspect pc adv) (current-static-aspects))
          body))]))

  (define-syntax (lambda/static stx)
    (syntax-case stx ()
      [(_ params body ...)
       (syntax
        (let ([aspects (current-static-aspects)])
          (λ params
            (w-c-m 'static-aspects aspects
              (begin body ...)))))]))

  (define (current-static-aspects)
    (let ([aspectss (c-c-m 'static-aspects)])
      (if (empty? aspectss)
          '()
          (first aspectss))))

  ;; redefine to incorporate both kinds
  (define (current-aspects)
    (append (current-dynamic-aspects)
            (current-static-aspects)))

  (provide (all-from-except mzscheme #%app lambda)
           ;;previous dynamic aspect part elided
           (rename lambda/static lambda)
           around))
```

Fig. 5. Extending Scheme with statically-scoped aspects

The definitions given in Figures 4 and 5 provide an implementation of aspect-scheme for statically- and dynamically-scoped aspects, with base language PLT Scheme. They are executable PLT Scheme code that correctly interprets the examples given in this paper. This code is available for download from the PLaneT package repository as AspectScheme 1.1 [11]. [9]

[9] AspectScheme programmers must preface their code with the following line:

```
(require (planet "aspect-scheme.ss" ("cdutchyn" "aspect-scheme.plt" 1 1)))
```

This automatically downloads, installs, and activates the language module.

4.4 Enhancements

Our implementation is Spartan in comparison with other AOP languages such as AspectJ. As shown above, it offers the necessary and sufficient features to provide dynamic aspect-oriented programming. There is also an enhanced version (AspectScheme 2.0 [12]) that mitigates several pedagogically-motivated design decisions. It

(1) extends pointcuts to access arguments other than the top-level join point: the idea is to incorporate arguments into the continuation mark. This removes the limitation described on page 9.
(2) distinguishes procedure call and execution join points, which becomes significant given access to arguments other than the top-level join point. Whereas call join points reflect procedure applications that have not yet started, execution join points (are presumed to) have begun. Hence, proceeding on call join points permits substitution of new argument values; proceeding on execution join points does not. Subtleties such as how advice can yield multiple active call join points arise.
(3) provides for top-level-scoped aspects; that is, advice that applies to all matching join points following injection into the REPL.
(4) recognizes advice-execution join points—places where advice has been activated by *app/weave/rt* instead of being manually applied to the original procedure to yield a new definition, as described in Costanza's non-oblivious dynamically-scoped functions [7].

This enhanced version, AspectScheme 2.0 is available from the PLaneT Package Repository also. [10]

4.5 Efficiency

Because we (intentionally) destroy tail-call optimization, our approach suffers from a run-time penalty. Given that **cflow** and **cflowbelow** pointcuts can discriminate the number and order of calls, it is straightforward to see that this cannot be improved to full tail-call optimization. In languages like AspectJ (sans **args** except for the top-most join point), the entire range of interesting continuation-mark sequences is known in advance. In that case, a regular automaton can recognize the join points [33], and the actual continuation marks need only denote the current automaton state. Thus, phased implementations

[10] To program in AspectScheme 2.0, use the prelude

(*require* (*planet* "aspect-scheme2.ss" ("cdutchyn" "aspect-scheme.plt" 2 0)))

to automatically download, install, and activate the enhanced language.

like AspectJ can restore tail-recursive optimizations for procedure calls which do not alter the automaton state; the process is similar to that described by Clemens and Felleisen [3, 4].

AspectScheme does not insist that aspects be predefined, so this technique has limited applicability. Consider the following example:

```
(let ([f (λ (g)
          (let ([h (λ () 1)])
            (around (&& (call h)
                        (cflow (call g)))
                    (λ (jp)
                      (λ ()
                        (+ 1 (jp))))
                    (h))))])
  (f f))
```

The application of f to itself makes the *cflow* (*call* g) pointcut true. Therefore, the aspect within f should be applied, requiring us to remember the calling context even in the absence of any aspects at the time of the call to f. AspectScheme can't know what calling context must be maintained in advance, so we must maintain the entire calling context.

In AspectScheme 2, because we support capturing arguments from cflow join points, a regular automaton no longer suffices: a push-down automaton is required. Now, there clearly is context building up and tail-call optimization is manifestly impossible in the general case. It is interesting to consider the cases where tail-call optimization remains possible.

Our implementation differs from other aspect languages, such as AspectJ, in another way. We present a dynamic weaving approach, but others resemble a code generation technique. This distinction is an artifact of AspectJ's implementation because the host language, Java, makes this straightforward and efficient. AspectJ's semantic description is dynamic as well [30]. Java divides execution into two separate phases: class elaboration and then expression evaluation. [11] Further, Java offers a flat scoping model and first-order definitions. Scheme offers richer lexical scoping, higher-order definitions, and explicitly interleaved phases (macro expansion, then expression evaluation) making program analysis of aspect code challenging.

[11] Dynamic class loading and load-time weaving may temporally interleave the phases, but they remain logically distinct.

5 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 machine model which we use to define our operational semantics. Subsequent sections then explain some key rules, including those for declaring aspects, testing function equality, and applying functions. Last, we informally sketch the connection between the implementation and the operational semantics.

5.1 Background on the CEKS *machine*

We use a variation on the CEKS machine [14] as the model for our semantics.

We have three reasons for using the CEKS machine in defining our semantics. First, recall that pointcuts can require knowledge of the control path that led to 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 a model that contains an abstract store.

The CEKS 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:

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

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 (+ 2 3):

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

We can represent this continuation as a tagged list:

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

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 5, the triple would be:

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

In order to use a CEKS machine, we must describe how to initiate a computation, and how recognize when it has terminated. Given a program, a closed top-level expression M, the machine is initialized with the triple:

$$\langle\langle M, E_0\rangle, \textsf{mt-k}, S_0\rangle$$

where $E_0 \equiv x \mapsto$ error is the initial environment that binds no variables, and $S_0 \equiv \ell \mapsto$ error is the initial store binding no locations. The machine steps through transitions until a terminal state $\langle\langle V, E\rangle, \textsf{mt-k}, S\rangle$ is reached, whereupon V is the final value of the program.

During the execution of the CEKS machine, various primitive operations must be performed. In our case, we provide a minimal sufficient set for manipulating the list values we support: *empty?*, *cons*, *first*, and *rest*. The transition rules

$$\begin{array}{l}\langle\langle(o\ M_1 \ldots M_n), E\rangle, K, S\rangle \\ \qquad \Rightarrow_{prim} \langle\langle M_1, E\rangle, \langle \textsf{op-k}\ o, \langle\rangle, \langle\langle M_2, E\rangle, \ldots, \langle M_n, E\rangle\rangle, K\rangle, S\rangle\end{array}$$

$$\begin{array}{l}\langle VC_m, \langle \textsf{op-k}\ o, \langle VC_{m-1}, \ldots, VC_1\rangle, \langle MC_{m+1}, \ldots, MC_n\rangle, K\rangle, S\rangle \\ \qquad \Rightarrow_{prim} \langle MC_{m+1}, \langle \textsf{op-k}\ o, \langle VC_m, \ldots, VC_1\rangle, \langle MC_{m+2}, \ldots, MC_n\rangle, K\rangle, S\rangle\end{array}$$

$$\langle VC_n, \langle \textsf{op-k}\ o, \langle VC_{n-1}, \ldots, VC_1\rangle, \langle\rangle, K\rangle, S\rangle \Rightarrow_{prim} \langle \delta(o, VC_1, \ldots, VC_n), K, S\rangle$$

where $VC = \langle V, E\rangle$ represents a closure of a value over an environment, illustrate that operands are evaluated left-to-right. The δ function receives the resulting value closures, and implements the actual primitives. Specifically, we define δ as

$$\begin{aligned}
\delta(\mathit{empty?}, VC) &= \langle \mathsf{true}, E_0 \rangle \text{ if } VC = \langle \mathsf{empty}, E \rangle \\
&= \langle \mathsf{false}, E_0 \rangle \text{ otherwise}
\end{aligned}$$

$$\delta(\mathit{cons}, VC_1, VC_2) = \langle (\mathsf{cons}\ \ VC_1\ \ VC_2), E_0 \rangle$$

$$\delta(\mathit{first}, \langle (\mathsf{cons}\ \ VC_1\ \ VC_2), E \rangle) = VC_1$$

$$\delta(\mathit{rest}, \langle (\mathsf{cons}\ \ VC_1\ \ VC_2), E \rangle) = VC_2$$

5.2 Declaring aspects

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

(**around** *pc adv body*)

(**fluid-around** *pc adv body*)

We will first describe the semantics of **around**; the semantics of **fluid-around** is 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 **lambda** 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_i, in closures:

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

where $\langle C, E, A\rangle$ is either a value closure $(C = V)$, abbreviated as VC, or an expression closure $(C = M)$, abbreviated as MC, and $A_0 \equiv \emptyset$ provides an initial, empty aspect environment.

The evaluation of **around** has three reduction rules. The first rule moves evaluation to the pointcut, M_{pc}, while remembering that the declaration was for a `static` aspect:

$$\begin{aligned}&\langle\langle(\textbf{around}\ M_{\mathrm{pc}}\ M_{\mathrm{adv}}\ M), E, A\rangle, K, S\rangle\\ &\qquad\Rightarrow_{\mathit{around}}\ \langle\langle M_{\mathrm{pc}}, E, A\rangle, \langle\textsf{around1-k}\ \texttt{static}, \langle M_{\mathrm{adv}}, E, A\rangle, \langle M, E, A\rangle, K\rangle, S\rangle\end{aligned}$$

The second rule says that once the pointcut computes to a value (VC_{pc}), evaluate the advice (MC_{adv}) next:

$$\begin{aligned}&\langle VC_{\mathrm{pc}}, \langle\textsf{around1-k}\ \mathit{scope}, MC_{\mathrm{adv}}, MC, K\rangle, S\rangle\\ &\qquad\Rightarrow_{\mathit{around}}\ \langle MC_{\mathrm{adv}}, \langle\textsf{around2-k}\ \mathit{scope}, VC_{\mathrm{pc}}, MC, K\rangle, S\rangle\end{aligned}$$

The third rule applies after both the pointcut and advice become values. The rule moves evaluation to the body of the **around** expression, but with an extended aspect environment. We add the triple $\langle \mathit{scope}, VC_{\mathrm{pc}}, VC_{\mathrm{adv}}\rangle$ to the aspect environment; that is, the scope tag (`static` for **around**), the pointcut value (VC_{pc}), and the advice value (VC_{adv}):

$$\begin{aligned}&\langle VC_{\mathrm{adv}}, \langle\textsf{around2-k}\ \mathit{scope}, VC_{\mathrm{pc}}, \langle M, E, A\rangle, K\rangle, S\rangle\\ &\qquad\Rightarrow_{\mathit{around}}\ \langle\langle M, E, A \cup \{\langle \mathit{scope}, VC_{\mathrm{pc}}, VC_{\mathrm{adv}}\rangle\}\rangle, K, S\rangle\end{aligned}$$

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

$$\begin{aligned}&\langle\langle(\textbf{fluid-around}\ M_{\mathrm{pc}}\ M_{\mathrm{adv}}\ M), E, A\rangle, K, S\rangle\\ &\qquad\Rightarrow_{\mathit{around}}\ \langle\langle M_{\mathrm{pc}}, E, A\rangle, \langle\textsf{around1-k}\ \texttt{dynamic}, \langle M_{\mathrm{adv}}, E, A\rangle, \langle M, E, A\rangle, K\rangle, S\rangle\end{aligned}$$

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

5.3 Function equality

Next we address the issue of function identity in a higher-order language. Recall that the pointcut of an aspect can refer to one or more procedures; for example, the pointcut (*call open-file*) denotes join points representing calls to the function *open-file*. Thus, at each function application, we must determine whether the function being applied is *open-file*. In a language like Java, this would be an easy test—we just use string equality to compare the name *open-file* 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 evaluates to a function. Second, even if the function *is* the variable *open-file*, we cannot tell by its name whether this was the *open-file* in scope when the aspect was defined. Consider the following expression:

```
(let ([open-file (λ (f) ...)])
  (around (call open-file) trace-advice
    (let ([open-file (λ (f) ...)])
      (open-file "vancouver"))))
```

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

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 **lambda** expression in the source program is labeled with a unique location identifier and each environment is labeled with a unique store location when it is constructed. In order to do this, we must extend our definition of environment to include a store location tag, $E :: \langle \ell, x \mapsto \ell \rangle$ with $E_0 = \langle \ell_0, x \mapsto \text{error} \rangle$, and store to include the set of locations allocated to environments, $S :: \langle \{\ell\}, \ell \mapsto VC \rangle$ with $S_0 = \langle \{\ell_0\}, \ell \mapsto \text{error} \rangle$ where ℓ_0 is initially allocated to E_0. This construction is similar to that given for R^{5}RS Scheme [22] in order to meet a minimal specification for *eq?*.

Two function closures are then *eq?* if and only if both location identifiers are the same and both environment locations are equal. The following case in the δ function illustrates this definition:

$$\begin{aligned}\delta(\mathit{eq?}, \langle (\lambda\,(x)\ M)_t, \langle \ell, e \rangle, A \rangle, \langle (\lambda\,(x')\ M')_{t'}, \langle \ell', e' \rangle, A' \rangle)&\\ &= \langle \mathsf{true}, E_0, A_0 \rangle \text{ if } t = t' \text{ and } \ell = \ell'\\ &= \langle \mathsf{false}, E_0, A_0 \rangle \text{ otherwise}\end{aligned}$$

This definition does not identify all functions that are observationally equal, but it is a conservative approximation of that relation that is both useful and can be computed in constant time.

5.4 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 from the site of application.

The evaluation of primitive application comprises three reduction rules. The first rule moves evaluation to the function position, M_{fun}, and keeps track of the static aspects from the aspect environment at the application site:

$$\begin{aligned}&\langle\langle(\textbf{app/prim}\ M_{\text{fun}}\ M_{\text{arg}}), E, A\rangle, K, S\rangle\\ &\qquad\Rightarrow_{\textit{app/prim}}\ \langle\langle M_{\text{fun}}, E, A\rangle, \langle \textsf{appprim1-k}\ \langle M_{\text{arg}}, E, A\rangle, A, K\rangle, S\rangle\end{aligned}$$

The second rule moves evaluation to the argument position, once the function is fully evaluated:

$$\begin{aligned}&\langle VC_{\text{fun}}, \langle\textsf{appprim1-k}\ MC_{\text{arg}}, A_{\text{app}}, K\rangle, S\rangle\\ &\qquad\Rightarrow_{\textit{app/prim}}\ \langle MC_{\text{arg}}, \langle\textsf{appprim2-k}\ VC_{\text{fun}}, A_{\text{app}}, K\rangle, S\rangle\end{aligned}$$

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

$$\begin{aligned}&\langle VC_{\text{arg}}, \langle\textsf{appprim2-k}\ \langle(\lambda\,(x)\ M)_t, E, A_{\text{fun}}\rangle, A_{\text{app}}, K\rangle, S\rangle\\ &\qquad\Rightarrow_{\textit{app/prim}}\ \langle\langle M, E', A'\rangle, K, S'\rangle\end{aligned}$$

where

$$\begin{aligned}\langle E', S'\rangle &= \langle E, S\rangle + \{x \mapsto VC_{\text{arg}}\}\\ A' &= A_{\text{app}}|_{\texttt{dynamic}} \cup A_{\text{fun}}|_{\texttt{static}}\end{aligned}$$

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

$$\langle\langle \ell, e\rangle, \langle L, s\rangle\rangle + \{x \mapsto VC\} \equiv \langle\langle \ell_e, e[x \mapsto \ell_v]\rangle, \langle L \cup \{\ell_e\}, s[\ell_v \mapsto VC]\rangle\rangle$$
$$\text{where } \ell_e, \ell_v \notin L \cup \text{dom}(S)$$

where $e[x \mapsto \ell]$ and $s[\ell \mapsto VC]$ employ the usual function extension.

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.

5.5 Regular function application

Three transition rules dictate the evaluation of function application. The first two steps are standard, because we do not invoke advice until the function and its argument are evaluated. The first rule moves evaluation to the function position, remembering the aspect environment from the application site:

$$\begin{aligned}&\langle\langle (M_{\text{fun}}\ M_{\text{arg}}), E, A\rangle, K, S\rangle\\ &\qquad \Rightarrow_{app} \langle\langle M_{\text{fun}}, E, A\rangle, \langle \mathsf{app1\text{-}k}\ \langle M_{\text{arg}}, E, A\rangle, E, A, K\rangle, S\rangle\end{aligned}$$

The second rule moves evaluation to the argument position:

$$\begin{aligned}&\langle VC_{\text{fun}}, \langle \mathsf{app1\text{-}k}\ MC_{\text{arg}}, E_{\text{app}}, A_{\text{app}}, K\rangle, S\rangle\\ &\qquad \Rightarrow_{app} \langle MC_{\text{arg}}, \langle \mathsf{app2\text{-}k}\ VC_{\text{fun}}, E_{\text{app}}, A_{\text{app}}, K\rangle, S\rangle\end{aligned}$$

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

Three things must happen during aspect invocation. First, we must generate a join point representing the application; second, we must test and apply any advice transforming the join point; and third, we must allow the transformed join point to execute.

We first examine join point generation. Recall that we represent a join point as a list of procedures, beginning with the one about to be applied, VC_{fun}, followed by the functions which are still in progress. In order for the continuation to make the "in-progress" functions available, we apply a special "application mark" (markapp-k) to the current continuation, which stores the called procedure. Given the existing continuation, K, we construct the new continuation

$K' = \langle \textsf{markapp-k}\ VC_{\text{fun}}, K\rangle$. It is important to note that this mark has no direct effect on the computation; when the function application returns, the following rule discards the mark:

$$\langle VC, \langle \textsf{markapp-k}\ VC_{\text{fun}}, K\rangle, S\rangle \Rightarrow_{mark} \langle VC, K, S\rangle$$

But, with this new continuation, we are now in position to compute the current join point which each pointcut takes as an argument. We simply traverse the continuation, cons-ing up the functions from the (pre-existing) continuation marks into a list:

$$\begin{aligned}
J[\![\textsf{mt-k}]\!] &= \langle \textsf{empty}, E_0, A_0\rangle \\
J[\![\langle \textsf{markapp-k}\ VC_{\text{fun}}, K\rangle]\!] &= \langle (\textsf{cons}\ VC_{\text{fun}},\ J[\![K]\!]), E_0, A_0\rangle \\
J[\![\langle \ldots, K\rangle]\!] &= J[\![K]\!] \text{ otherwise}
\end{aligned}$$

Second, we must check and apply each aspect in the aspect environment,

$$A = \{\langle \mathit{scope}, pc^i, adv^i\rangle \mid i = 1, \ldots, \mid A \mid\}.$$

This entails applying pc^i to the join point in context ($jp*$). If this returns true, then we applying the corresponding advice adv^i (a procedure transformer) to the function to yield a new function. Otherwise, we return the original (untransformed) procedure. The transformation, W, with base case of the original function, *fun*, is given by:

$$\begin{aligned}
W[\![0]\!] &= \mathit{fun} \\
W[\![i]\!] &= (\textbf{app/prim}\ (\lambda\,(f)\ (\textbf{if}\ (\textbf{app/prim}\ pc^i\ jp*) \\
&\qquad\qquad\qquad\qquad (\textbf{app/prim}\ adv^i\ f) \\
&\qquad\qquad\qquad\qquad f)) \\
&\qquad\quad W[\![i-1]\!]) \qquad\qquad \text{for } i > 0
\end{aligned}$$

Notice the following points about W:

(1) If no advice exists, it simply returns the original function, which will be applied, using **app/prim**, to the argument.
(2) It applies each pointcut to the join point.
(3) If no pointcut holds, again it returns the original function.
(4) If some pointcuts hold, then it uses **app/prim** to apply the final transformed procedure to the original argument. Note that applications in the body of the transformed procedure may also invoke aspects.

Third, we take the procedure resulting from all applicable advice transformations and **app/prim** it to the original argument, yielding a new expression for this function application:

$$M' = (\textbf{app/prim}\ W[\![|\ A_{\text{app}}\ |]\!]\ \mathit{arg})$$

The third transition rule applies aspects as described above, binding the various variables for the function, *fun*, the argument, *arg*, join point, *jp*∗, and all aspect components (pc^i and adv^i) in the environment and store. Evaluation moves to the new M', carrying the dynamic aspect environment A_{app} for use within M', but the static aspect environment remains available as part of the *fun* closure.

$$\begin{aligned}
&\langle VC_{\text{arg}}, \langle \textsf{app2-k}\ \langle(\lambda\,(x)\ M)_t, E_{\text{fun}}, A_{\text{fun}}\rangle, E_{\text{app}}, A_{\text{app}}, K\rangle, S\rangle \\
&\quad \Rightarrow_{\mathit{app}}\ \langle\langle M', E', A_{\text{app}}\rangle, K', S'\rangle \\
&\quad \text{where} \\
&\qquad M' = (\textbf{app/prim}\ W[\![|\ A_{\text{app}}\ |]\!]\ \mathit{arg}) \\
&\qquad K' = \langle \textsf{markapp-k}\ \langle(\lambda\,(x)\ M)_t, E_{\text{fun}}, A_{\text{fun}}\rangle, K\rangle \\
&\quad \langle E', S'\rangle = \langle E_{\text{app}}, S\rangle \\
&\qquad\quad + \{\mathit{fun} \mapsto \langle(\lambda\,(x)\ M)_t, E_{\text{fun}}, A_{\text{fun}}\rangle,\ \mathit{arg} \mapsto VC_{\text{arg}},\ \mathit{jp}{*} \mapsto J[\![K']\!]\} \\
&\qquad\quad + \{\mathit{pc}^i \mapsto VC_{\text{pc}^i}, \mathit{adv}^i \mapsto VC_{\text{adv}^i} \mid \langle \mathit{scope}, VC_{\text{pc}^i}, VC_{\text{adv}^i}\rangle \in A_{\text{app}}\}
\end{aligned}$$

5.6 From Semantics to Implementation

We now sketch the correspondence between the semantic specification and our AspectScheme implementation. To see the connection, we need to examine four specific constructions that provide our aspect-oriented behavior.

First, we recognize that in each case the weaver is implemented as a procedure transformer. *Weave* is equivalent to W, transforming a procedure by testing pointcuts and applying advice; and differs only in the data representation recursed over. *Weave* recurs, via *foldr*, over the list of applicable aspects given by *current-aspects*; whereas W recurs over the size of the set A_{app}. This difference avoids cluttering the semantics with list operations. Our *app/weave* macro intercepts procedure applications in Scheme and inserts aspect behavior into the execution via *app/weave/rt*. The semantics distinguishes primitive and procedure applications with separate *prim* and *app* transition rules, and *app/weave/rt* captures this distinction. In the former case, *app/weave/rt* allows the primitive procedure to continue without alteration. In the latter case, it applies the results of *weave* to the procedure arguments exactly as the *app* transitions specify.

Second, we recognize that our implementation of **w-c-m** builds new continuation frames that apply the identity procedure. As continuation frames that simply pass values through, these serve as fresh placeholders to contain the keyed mark which **w-c-m** writes. This behavior matches that of markapp-k continuation frames in our semantics. In particular, our implemented continuation marks keyed by 'joinpoint contain applied procedures, paralleling our markapp-k continuation frames storing procedure values. Our **c-c-m** implementation gathers all continuation marks for a given key; for the 'joinpoint key, this yields the result of J in the semantics.

Third, we observe that dynamic aspects form a subset of the dynamically-scoped aspect environment A in the operational semantics, being those elements with *scope* = `dynamic`. **W-c-m** offers exactly that same dynamic scoping, but allows us to differentiate dynamic from static aspects by key. Hence, implementing dynamic aspects as continuation marks keyed with 'dynamic-aspect allows **c-c-m** to gather all dynamic aspects still on the stack, yielding (*current-dynamic-aspects*), which corresponds to $A_{\text{app}}|_{\texttt{dynamic}}$ in the transition rule at the bottom of page 35).

Fourth, we note that the operational semantics accumulates static aspects in the dynamically scoped variable A (with *scope* = `static`). The operational semantics specifies that evaluation of λ must close over A at definition time and apply only the static subset at application time (corresponding to $A_{\text{fun}}|_{\texttt{static}}$ in the transition rule at the bottom of page 35). Inspection shows that our **lambda/static** does precisely that: it captures the definition-time static aspects into a lexical variable, *aspects*, and reinstates them at the start of the procedure body for use by the weaver at application time.

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

6 Related Work

While the earlier work on aspects [24] 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 [23] 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 pointcuts, 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. This paper, however, describes various shortcomings of AspectJ relative to our language.

Wand, Kiczales, and Dutchyn [37] 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.

Walker, Zdancewic, and Ligatti [36] offer an operational account of aspect-oriented programming, based on translation to a core language enriched with labels. In many ways, their core language follows our approach: labels act like continuation marks, advice is captured in an environment, advice is procedure composition, and pointcuts are predicates over machine state represented by marks. The key difference is that we allow direct access to the pointcuts and advice, making them programmable and higher-order; whereas Walker et al. fix them in a translation from the external application programming language. The enables us to recognize static scoping of aspects, a feature not supported in their system. Dantas et al. [8, 9] continue Walker et al.'s approach, extending it with polymorphic type-checking and local type inference.

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.

Douence, Motelet, and Südholt [10] provide one of the earliest formal descriptions of AOP. They frame aspect-oriented programming as program execution monitoring, where *crosscuts* (now called pointcuts) match events in context. Their work previews our first-class predicate formulation of pointcuts, by providing combinators for building up pointcuts. But they keep the aspects as separate execution monitoring code. They rewrite an original program to insert calls to that separate monitoring code, so that the monitoring code can recognize pointcuts, execute advice, and pass control back. This substantial difference results in advice that is static and top-level only in their system.

Orleans [32] defines a predicate dispatching system for Scheme, where the system dispatches each message according to a *branch* consisting of a predicate and a body. These two components are first class values, and thus equivalent to our pointcut 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.

Lämmel and Visser [28] offer a number of *functional strategies* for traversing and transforming abstract syntaxes. Their traversal strategies are reminiscent of our higher-order pointcuts, providing very expressive ways of declaring the target and context of a transformation. But, their work differs from ours in that they apply syntactic rewriting to analyze and transform programs whereas our system operates at evaluation time with full access to values.

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 [37] is especially valuable, because it makes clear the essence of these kinds of aspects, making it easier to port this style of programming between languages. Because that 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 Ruby 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 designers 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. On 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 akin to those of AspectJ and static aspects that can enforce policies defined within modules, e.g., 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 [17]. This makes it very convenient for programmers to exploit aspects to improve program designs without changing their program development methodology. In addition, continuation marks impose low run-time overhead, so programmers are not penalized for their use. Finally, we give a semantic description of AspectScheme, providing a firm foundation for understanding, verifying and extending aspects.

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. Also, we might use parametricity to define more general aspects, and develop aspect combinators by employing higher-order functions. 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.

Acknowledgments We acknowledge John Clements for his valuable comments on preliminary versions of this paper. Thanks also to Matthias Felleisen and John Clements for discussions on this research. We are grateful to Gregor Kiczales for his support and encouragement. We thank Clint Morgan for reporting on his experience with AspectScheme 2.0 in a course project. The anonymous reviewers provided detailed responses that immensely improved the content and presentation of this material.

References

[1] L. Bauer, J. Ligatti, D. Walker, A calculus for composing security policies, Tech. Rep. TR-655-02, Princeton University (2002).

[2] L. Bergmans, M. Aksit, Composing crosscutting concerns using composition filters, Communications of the ACM 44 (10) (2001) 51–57.

[3] J. Clements, M. Felleisen, A tail-recursive semantics for stack inspections, in: European Symposium on Programming, Lecture Notes in Computer Science 2618 (2003) 22–37.

[4] J. Clements, M. Felleisen, A tail-recursive machine with stack inspection, ACM Transactions on Programming Languages and Systems 26 (6) (2004) 1029–1052.

[5] J. Clements, M. Flatt, M. Felleisen, Modeling an algebraic stepper, in: European Symposium on Programming, Lecture Notes in Computer Science 2029 (2001) 320–334.

[6] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.

[7] P. Costanza, Dynamically scoped functions as the essence of AOP, in: Workshop on Object-Oriented Language Engineering for the Post-Java Era, ACM SIGPLAN Notices 38 (8) (2003) 27–36.

[8] D. S. Dantas, D. Walker, G. Washburn, S. Weirich, Analyzing polymorphic advice, Tech. Rep. TR-717-05, Princeton University (2005).

[9] D. S. Dantas, D. Walker, G. Washburn, S. Weirich, PolyAML: a polymorphic aspect-oriented functional programmming language, in: ACM International Conference on Functional Programming, 2005.

[10] R. Douence, O. Motelet, M. Südholt, A formal definition of crosscuts, in: International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Lecture Notes in Computer Science 2192 (2001) 170–186.

[11] C. Dutchyn, AspectScheme 1.1, `http://www.cs.ubc.ca/cdutchyn-/downloads/AspectScheme/aspectscheme1-1.plt` (Jan. 2005).

[12] C. Dutchyn, AspectScheme 2.0, `http://www.cs.ubc.ca/cdutchyn-/downloads/AspectScheme/aspectscheme2-2.plt` (Oct. 2005).

[13] R. K. Dybvig, R. Hieb, C. Bruggeman, Syntactic abstraction in Scheme, Lisp and Symbolic Computation 5 (4) (1993) 295–326.

[14] M. Felleisen, R. Hieb, The revised report on the syntactic theories of sequential control and state, Theoretical Computer Science 103 (2) (1992) 235–271.

[15] R. E. Filman, What is aspect-oriented programming, revisited, Workshop on Advanced Separation of Concerns, ECOOP, also RIACS Tech. Rep. 01.14, 2001.

[16] R. E. Filman, D. P. Friedman, Aspect-oriented programming is quantification and obliviousness, in: Workshop on Advanced Separation of Concerns, OOPSLA, 2000, also RIACS Tech. Rep. 01.12, 2001.

[17] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, M. Felleisen, DrScheme: A programming environment for Scheme, Journal of Functional Programming 12 (2) (2002) 159–182.

[18] M. Flatt, Composable and compilable macros: You want it when?, in: ACM International Conference on Functional Programming, 2002, 72–83.

[19] M. Flatt, PLT MzScheme: Language manual, Tech. Rep. TR97-280, Rice University (1997).

[20] J. Gray, T. Bapty, S. Neema, J. Tuck, Handling crosscutting constraints in domain-specific modeling, Communications of the ACM 44 (10) (2001) 87–93.

[21] C. Hanson, the MIT Scheme team, and others, MIT Scheme reference manual, Massachusetts Institute of Technology, 2002.

[22] R. Kelsey, W. Clinger, J. Rees, Revised^5 report on the algorithmic language Scheme, Higher-Order and Symbolic Computation 11 (1) (1998), 7–105.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, in: European Conference on Object-Oriented Programming, Lecture Notes in Computer Science 2072 (2001) 327–353.

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: European Conference on Object-Oriented Programming, Lecture Notes in Computer Science 1241 (1997) 220–242.

[25] E. E. Kohlbecker Jr, Syntactic extensions in the programming language Lisp, Ph.D. thesis, Indiana University (Aug. 1986).

[26] E. E. Kohlbecker Jr, D. P. Friedman, M. Felleisen, B. F. Duba, Hygienic macro expansion, in: ACM Conference on Lisp and Functional Programming, 1986, 151–161.

[27] E. E. Kohlbecker, M. Wand, Macros-by-example: Deriving syntactic transformations from their specifications, in: ACM Annual Symposium on Principles of Programming Languages, 1987, 77–84.

[28] R. Lämmel, J. Visser, Design patterns for functional strategic programming, in: ACM Workshop on Rule-Based Programming, 2002.

[29] K. Lieberherr, D. Orleans, J. Ovlinger, Aspect-oriented programming with adaptive methods, Communications of the ACM 44 (10) (2001) 39–41.

[30] H. Masuhara, G. Kiczales, C. Dutchyn, A compilation and optimization model for aspect-oriented programs, in: Compiler Construction, Lecture Notes in Computer Science 2622 (2003) 46–60.

[31] P. Netinant, T. Elrad, M. 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) (2001) 83–85.

[32] D. Orleans, Incremental programming with extensible decisions, in: ACM Conference on Aspect-Oriented Software Development, 2002, 56–64.

[33] D. Sereni, O. de Moor, Static analysis of aspects, in: ACM Conference on Aspect-Oriented Software Development, 2003, 30–39.

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

[35] D. B. Tucker, S. Krishnamurthi, Pointcuts and advice in higher-order languages, in: ACM Conference on Aspect-Oriented Software Development, 2003, 158–167.

[36] D. Walker, S. Zdancewic, J. Ligatti, A theory of aspects, in: ACM International Conference on Functional Programming, 2003, 127–139.

[37] M. Wand, G. Kiczales, C. Dutchyn, A semantics for advice and dynamic join points in aspect-oriented programming, ACM Transactions on Programming Languages and Systems 26 (5) (2004) 890–910.

A Semantics

Values V ::= $(\lambda\,(x)\ M)_t$

| true

| false

| empty

| $(\mathsf{cons}\ VC\ VC)$

x :: identifier

t :: source location tag

Expressions M ::= V

| x

| $(M\ M)$

| $(o\ M\ \ldots)$

| $(\textbf{if}\ M\ M\ M)$

| $(\textbf{set!}\ x\ M)$

| $(\textbf{around}\ M\ M\ M)$

| $(\textbf{fluid-around}\ M\ M\ M)$

| $(\textbf{app/prim}\ M\ M)$

Primitive operations o ::= $\mathit{eq?}$

| cons

| first

| rest

| $\mathit{empty?}$

Stores S :: $\langle\{\ell\}, \ell \to VC\rangle$

$S_0 \equiv \langle\{\ell_0\}, \ell \mapsto \text{error}\rangle$

ℓ :: store location

ℓ_0 = fixed store location

Environments E :: $\langle \ell, x \to \ell\rangle$

$E_0 \equiv \langle \ell_0, x \mapsto \text{error}\rangle$

$$\langle\langle \ell, e\rangle, \langle L, s\rangle + \{x \mapsto VC\} \equiv \langle\langle \ell_e, e[x \mapsto \ell_v]\rangle, \langle L \cup \{\ell_e\}, s[\ell_v \mapsto VC]\rangle\rangle$$
$$\text{where } \ell_e, \ell_v \notin L \cup \text{dom}(S)$$

$$\langle E, S\rangle + \{x_1 \mapsto VC_1, \ldots, x_n \mapsto VC_n\}$$
$$\equiv \langle E, S\rangle + \{x_1 \mapsto VC_1\} + \cdots + \{x_n \mapsto VC_n\}$$

Aspect environments	A	$::$	$\{\langle \mathit{scope}, \mathit{VC}, \mathit{VC} \rangle\}$
	A_0	$\equiv$	$\emptyset$
			$\mathit{scope} \in \{\texttt{static}, \texttt{dynamic}\}$
Expression closures	MC	$::=$	$\langle M, E, A \rangle$
Value closures	VC	$::=$	$\langle V, E, A \rangle$
Continuations	K	$::=$	$\textsf{mt-k}$
		$\mid$	$\langle \textsf{app1-k}\ MC, E, A, K \rangle$
		$\mid$	$\langle \textsf{app2-k}\ VC, E, A, K \rangle$
		$\mid$	$\langle \textsf{if-k}\ MC, MC, K \rangle$
		$\mid$	$\langle \textsf{set-k}\ x, E, A, K \rangle$
		$\mid$	$\langle \textsf{around1-k}\ \mathit{scope}, MC, MC, K \rangle$
		$\mid$	$\langle \textsf{around2-k}\ \mathit{scope}, VC, MC, K \rangle$
		$\mid$	$\langle \textsf{markapp-k}\ VC, K \rangle$
		$\mid$	$\langle \textsf{appprim1-k}\ MC, A, K \rangle$
		$\mid$	$\langle \textsf{appprim2-k}\ VC, A, K \rangle$
		$\mid$	$\langle \textsf{op-k}\ o, \langle VC, \ldots \rangle, \langle MC, \ldots \rangle, K \rangle$

intialization and termination

$$M \ \Rightarrow_{\mathit{init}} \ \langle \langle M, E_0, A_0 \rangle, \textsf{mt-k}, S_0 \rangle$$

$$\langle \langle V, E, A \rangle, \textsf{mt-k}, S \rangle \Rightarrow_{\mathit{term}} V$$

variables

$$\langle \langle x, E, A \rangle, K, S \rangle \ \Rightarrow_{\mathit{var}} \ \langle S(E(x)), K, S \rangle$$

if

$$\langle \langle (\textbf{if}\ M\ \ M_{\text{then}}\ \ M_{\text{else}}), E, A \rangle, K, S \rangle$$
$$\Rightarrow_{\mathit{if}} \ \langle \langle M, E, A \rangle, \langle \textsf{if-k}\ \langle M_{\text{then}}, E, A \rangle, \langle M_{\text{else}}, E, A \rangle, K \rangle, S \rangle$$

$$\langle \langle \textsf{true}, E, A \rangle, \langle \textsf{if-k}\ MC_{\text{then}}, MC_{\text{else}}, K \rangle, S \rangle \ \Rightarrow_{\mathit{if}} \ \langle MC_{\text{then}}, K, S \rangle$$

$$\langle \langle \textsf{false}, E, A \rangle, \langle \textsf{if-k}\ MC_{\text{then}}, MC_{\text{else}}, K \rangle, S \rangle \ \Rightarrow_{\mathit{if}} \ \langle MC_{\text{else}}, K, S \rangle$$

continuation marks

$$\langle VC, \langle \textsf{markapp-k}\ VC_{\text{fun}}, K \rangle, S \rangle \ \Rightarrow_{\mathit{mark}} \ \langle VC, K, S \rangle$$

set!

$\langle\langle(\textbf{set!}\ x\ M), E, A\rangle, K, S\rangle \Rightarrow_{set} \langle\langle M, E, A\rangle, \langle \textsf{set-k}\, x, E, A, K\rangle, S\rangle$

$\langle VC, \langle \textsf{set-k}\, x, E, A, K\rangle, S\rangle \Rightarrow_{set} \langle VC, K, S[E(x) \mapsto VC]\rangle$

primitive operations

$\langle\langle(o\ M_1 \ldots M_n), E, A\rangle, K, S\rangle$
$\quad \Rightarrow_{prim} \langle\langle M_1, E, A\rangle, \langle \textsf{op-k}\, o, \langle\rangle, \langle\langle M_2, E, A\rangle, \ldots, \langle M_n, E, A\rangle\rangle, K\rangle, S\rangle$

$\langle VC_m, \langle \textsf{op-k}\, o, \langle VC_{m-1}, \ldots, VC_1\rangle, \langle MC_{m+1}, \ldots, MC_n\rangle, K\rangle, S\rangle$
$\quad \Rightarrow_{prim} \langle MC_{m+1}, \langle \textsf{op-k}\, o, \langle VC_m, \ldots, VC_1\rangle, \langle MC_{m+2}, \ldots, MC_n\rangle, K\rangle, S\rangle$

$\langle VC_n, \langle \textsf{op-k}\, o, \langle VC_{n-1}, \ldots, VC_1\rangle, \langle\rangle, K\rangle, S\rangle \Rightarrow_{prim} \langle \delta(o, VC_1, \ldots, VC_n), K, S\rangle$

where

$$\begin{aligned}\delta(\mathit{empty?}, VC) &= \langle \textsf{true}, E_0, A_0\rangle \text{ if } VC = \langle \textsf{empty}, E, A\rangle \\ &= \langle \textsf{false}, E_0, A_0\rangle \text{ otherwise}\end{aligned}$$

$$\delta(\mathit{cons}, VC_1, VC_2) = \langle(\textsf{cons}\ VC_1\ VC_2), E_0, A_0\rangle$$

$$\delta(\mathit{first}, \langle(\textsf{cons}\ VC_1\ VC_2), E, A\rangle) = VC_1$$

$$\delta(\mathit{rest}, \langle(\textsf{cons}\ VC_1\ VC_2), E, A\rangle) = VC_2$$

$$\begin{aligned}&\delta(\mathit{eq?}, \langle(\lambda\,(x)\ M)_t, \langle \ell, e\rangle, A\rangle, \langle(\lambda\,(x')\ M')_{t'}, \langle \ell', e'\rangle, A'\rangle) \\ &\qquad = \langle \textsf{true}, E_0, A_0\rangle \text{ if } t = t' \text{ and } \ell = \ell' \\ &\qquad = \langle \textsf{false}, E_0, A_0\rangle \text{ otherwise}\end{aligned}$$

around and **fluid-around**

$\langle\langle(\textbf{around}\ M_{\text{pc}}\ M_{\text{adv}}\ M), E, A\rangle, K, S\rangle$
$\quad \Rightarrow_{around} \langle\langle M_{\text{pc}}, E, A\rangle, \langle \textsf{around1-k}\ \texttt{static}, \langle M_{\text{adv}}, E, A\rangle, \langle M, E, A\rangle, K\rangle, S\rangle$

$\langle\langle(\textbf{fluid-around}\ M_{\text{pc}}\ M_{\text{adv}}\ M), E, A\rangle, K, S\rangle$
$\quad \Rightarrow_{around} \langle\langle M_{\text{pc}}, E, A\rangle, \langle \textsf{around1-k}\ \texttt{dynamic}, \langle M_{\text{adv}}, E, A\rangle, \langle M, E, A\rangle, K\rangle, S\rangle$

$\langle VC_{\text{pc}}, \langle \textsf{around1-k}\ \mathit{scope}, MC_{\text{adv}}, MC, K\rangle, S\rangle$
$\quad \Rightarrow_{around} \langle MC_{\text{adv}}, \langle \textsf{around2-k}\ \mathit{scope}, VC_{\text{pc}}, MC, K\rangle, S\rangle$

$\langle VC_{\text{adv}}, \langle \textsf{around2-k}\ \mathit{scope}, VC_{\text{pc}}, \langle M, E, A\rangle, K\rangle, S\rangle$
$\quad \Rightarrow_{around} \langle\langle M, E, A \cup \{\langle \mathit{scope}, VC_{\text{pc}}, VC_{\text{adv}}\rangle\}\rangle, K, S\rangle$

app/prim

$$\langle\langle(\textbf{app/prim}\ M_{\text{fun}}\ M_{\text{arg}}), E, A\rangle, K, S\rangle$$
$$\quad\Rightarrow_{\mathit{app/prim}}\ \langle\langle M_{\text{fun}}, E, A\rangle, \langle\textsf{appprim1-k}\ \langle M_{\text{arg}}, E, A\rangle, A, K\rangle, S\rangle$$

$$\langle VC_{\text{fun}}, \langle\textsf{appprim1-k}\ MC_{\text{arg}}, A_{\text{app}}, K\rangle, S\rangle$$
$$\quad\Rightarrow_{\mathit{app/prim}}\ \langle MC_{\text{arg}}, \langle\textsf{appprim2-k}\ VC_{\text{fun}}, A_{\text{app}}, K\rangle, S\rangle$$

$$\langle VC_{\text{arg}}, \langle\textsf{appprim2-k}\ \langle(\lambda\,(x)\ M)_t, E, A_{\text{fun}}\rangle, A_{\text{app}}, K\rangle, S\rangle$$
$$\quad\Rightarrow_{\mathit{app/prim}}\ \langle\langle M, E', A'\rangle, K, S'\rangle$$

where

$$\langle E', S'\rangle = \langle E, S\rangle + \{x \mapsto VC_{\text{arg}}\}$$
$$A' = A_{\text{app}}|_{\texttt{dynamic}} \cup A_{\text{fun}}|_{\texttt{static}}$$

function applications

$$\langle\langle(M_{\text{fun}}\ M_{\text{arg}}), E, A\rangle, K, S\rangle$$
$$\quad\Rightarrow_{\mathit{app}}\ \langle\langle M_{\text{fun}}, E, A\rangle, \langle\textsf{app1-k}\ \langle M_{\text{arg}}, E, A\rangle, E, A, K\rangle, S\rangle$$

$$\langle VC_{\text{fun}}, \langle\textsf{app1-k}\ MC_{\text{arg}}, E_{\text{app}}, A_{\text{app}}, K\rangle, S\rangle$$
$$\quad\Rightarrow_{\mathit{app}}\ \langle MC_{\text{arg}}, \langle\textsf{app2-k}\ VC_{\text{fun}}, E_{\text{app}}, A_{\text{app}}, K\rangle, S\rangle$$

$$\langle VC_{\text{arg}}, \langle\textsf{app2-k}\ \langle(\lambda\,(x)\ M)_t, E_{\text{fun}}, A_{\text{fun}}\rangle, E_{\text{app}}, A_{\text{app}}, K\rangle, S\rangle$$
$$\quad\Rightarrow_{\mathit{app}}\ \langle\langle M', E', A_{\text{app}}\rangle, K', S'\rangle$$

where

$$M' = (\textbf{app/prim}\ W[\![\,|\ A_{\text{app}}\ |\,]\!]\ \mathit{arg})$$

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

$$\begin{aligned}\langle E', S'\rangle = \langle E_{\text{app}}, S\rangle &\\ &+ \{\mathit{fun} \mapsto \langle(\lambda\,(x)\ M)_t, E_{\text{fun}}, A_{\text{fun}}\rangle,\ \mathit{arg} \mapsto VC_{\text{arg}},\ \mathit{jp*} \mapsto J[\![K']\!]\}\\ &+ \{pc^i \mapsto VC_{\text{pc}^i}, \mathit{adv}^i \mapsto VC_{\text{adv}^i} \mid \langle \mathit{scope}^i, VC_{\text{pc}^i}, VC_{\text{adv}^i}\rangle \in A_{\text{app}}\}\end{aligned}$$

$$\begin{aligned}W[\![0]\!] &= \mathit{fun}\\ W[\![i]\!] &= (\textbf{app/prim}\ (\lambda\,(f)\ (\textbf{if}\ (\textbf{app/prim}\ pc^i\ \mathit{jp*})\\ &\qquad\qquad\qquad\qquad (\textbf{app/prim}\ \mathit{adv}^i\ f)\\ &\qquad\qquad\qquad\qquad f))\\ &\qquad\qquad W[\![i-1]\!]) \qquad\qquad \text{for } i > 0\end{aligned}$$

$$\begin{aligned}J[\![\textsf{mt-k}]\!] &= \langle\textsf{empty}, E_0, A_0\rangle\\ J[\![\langle\textsf{markapp-k}\ VC, K\rangle]\!] &= \langle(\textsf{cons}\ VC\ J[\![K]\!]), E_0, A_0\rangle\\ J[\![\langle\ldots, K\rangle]\!] &= J[\![K]\!]\ \text{otherwise}\end{aligned}$$

B Implementation

```
;;;
;;; AspectScheme 1.1
;;;

(module aspect-scheme mzscheme
  (require (only (lib "list.ss") foldr first rest empty?))

  ;; continuation mark interface
  (define (c-c-m key)
    (continuation-mark-set→list
     (current-continuation-marks)
     key))

  (define-syntax (w-c-m stx)
    (syntax-case stx ()
      [(_ tag mark body)
       (syntax
         ((λ (x) x)
          (with-continuation-mark tag mark body)))]))

  ;; aspect structures
  (define-struct aspect (pc adv))

  ;; current aspects (both static and dynamic)
  (define (current-aspects)
    (append (current-dynamic-aspects)
            (current-static-aspects)))

  ;; dynamically-scoped aspects
  (define-syntax (fluid-around stx)
    (syntax-case stx ()
      [(_ pc adv body)
       (syntax (w-c-m 'dynamic-aspect (make-aspect pc adv) body))]))

  (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
        (apply (weave fun-val (c-c-m 'joinpoint) (current-aspects))
               arg-vals))))

(define (weave fun-val jp* aspects)
  (foldr (λ (a r)
            (if ((aspect-pc a) jp*)
                ((aspect-adv a) r)
                r))
          fun-val
          aspects))

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

;; statically-scoped aspects
(define-syntax (around stx)
  (syntax-case stx ()
    [(_ pc adv body)
     (syntax
       (w-c-m 'static-aspects
                (cons (make-aspect pc adv) (current-static-aspects))
         body))]))

(define-syntax (lambda/static stx)
  (syntax-case stx ()
    [(_ params body ...)
     (syntax
      (let ([aspects (current-static-aspects)])
        (λ params
          (w-c-m 'static-aspects aspects
            (begin body ...)))))]))

(define (current-static-aspects)
  (let ([aspects (c-c-m 'static-aspects)])
    (if (empty? aspects)
        '()
        (first aspects))))
```

```
;; pointcuts - fundamental
;; NB: app/prim is not required within this module because
;; the replacement of #%app with app/weave
;; does not occur until after the module is installed.
(define ((call f) jp*)
  (eq? f (first jp*)))

(define ((top) jp*)
  (empty? jp*))

(define ((below pc) jp*)
  (and (not (empty? (rest jp*)))
       (pc (rest jp*))))

;; pointcuts - strict combinators, variadic
(define ((&& . pcs) jp*)
  (andmap (λ (pc) (pc jp*)) pcs))

(define ((|| . pcs) jp*)
  (ormap (λ (pc) (pc jp*)) pcs))

(define ((! pc) jp*)
  (not (pc jp*)))

;; pointcuts - higher-order points-free
(define (within f)
  (below (call f)))

(define (cflow pc)
  (&& (! top)
      (|| pc (below (cflow pc)))))

(define (cflowbelow pc)
  (&& (! top)
      (below (cflow pc))))

(provide (all-from-except mzscheme #%app lambda)
         (rename app/weave #%app)
         (rename #%app app/prim)
         fluid-around
         (rename lambda/static lambda)
         around
         call top below && || ! within cflow cflowbelow))
```