A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming

MITCHELL WAND
Northeastern University
and
GREGOR KICZALES and CHRISTOPHER DUTCHYN
University of British Columbia

A characteristic of aspect-oriented programming, as embodied in AspectJ, is the use of advice and pointcuts to define behavior that crosscuts the structure of the rest of the code. The events during execution at which advice may execute are called *join points*. A *pointcut* is a set of join points. An advice is an action to be taken at the join points in a particular pointcut. In this model of aspect-oriented programming, join points are dynamic in that they refer to events during the flow of execution of the program.

We give a denotational semantics for a minilanguage that embodies the key features of dynamic join points, pointcuts, and advice. This is the first semantics for aspect-oriented programming that handles dynamic join points and recursive procedures. It is intended as a baseline semantics against which future correctness results may be measured.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Denotational semantics*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: Advice, AspectJ, aspect-oriented programming, Aspect Sand Box, join point, pointcut

Earlier versions of this article were presented at the 9th International Workshop on Foundations of Object-Oriented Languages, January 19, 2002, and at the Workshop on Foundations of Aspect-Oriented Languages (FOAL), April 22, 2002.
This work was supported in part by the U.S. National Science Foundation under grants CCR-9804115 and CCR-0097740, and by the Natural Science Research Council (NSERC) of Canada, Sierra Systems Group and Xerox Canada Limited.
Authors' addresses: M. Wand, College of Computer and Information Science, Northeastern University, 360 Huntington Avenue, Boston, MA 02115; email: m.wand@neu.edu; G. Kiczales and C. Dutchyn, Department of Computer Science, University of British Columbia, 201-2366 Main Mall, Vancouver, BC V6T 1Z4, Canada; email: {gregor,cdutchyn}@cs.ubc.ca.

1. INTRODUCTION

A characteristic of aspect-oriented programming, as embodied in AspectJ [Kiczales et al. 2001], is the use of advice and pointcuts to define behavior that crosscuts the structure of the rest of the code. The events during execution at which advice may execute are called *join points*. A *pointcut* is a set of join points. An *advice* is an action to be taken at the join points in a particular pointcut. In this model of aspect-oriented programming, join points are dynamic in that they refer to events during the flow of execution of the program. The process of causing the relevant advice at each join point to be executed is called *weaving*.

The condition is specified by a formula called a *pointcut designator* or *pcd*. A typical pcd might look like

```
(and (pcall f) (pwithin g) (cflow (pcall h))).
```

This indicates that the piece of advice to which this pcd is attached is to be executed at every call to procedure `f` from within the text of procedure `g`, but only when that call occurs dynamically within a call to procedure `h`.

This article presents a model of dynamic join points, pointcut designators, and advice. It introduces a tractable minilanguage embodying these features and gives it a denotational semantics.

This is the first semantics for aspect-oriented programming that handles dynamic join points and recursive procedures. It is intended as a baseline against which future correctness results may be measured.

This work is part of the Aspect Sand Box (ASB) project [Dutchyn et al. 2002; Masuhara and Kiczales 2003]. The goal of ASB is to produce an experimental workbench for aspect-oriented programming of various flavors. ASB includes a small base language and is intended to include a set of exemplars of different approaches to AOP. The work reported here is a model of one of those exemplars, namely dynamic join points and advice with dynamic weaving. ASB also includes other AOP models, including static join points, Demeter [Lieberherr 1996], and Hyper/J [Ossher and Tarr 2000], and both interpreter-like and compiler-like implementation models.

For more motivation for AOP, see Kiczales et al. [1997] or the articles in Elrad et al. [2001]. For more on AspectJ, see Kiczales et al. [2001].

2. A MODEL

We begin by presenting a conceptual model of aspect-oriented programming with dynamic join points as found in AspectJ.

In this model, a program consists of a base program and some pieces of *advice*. The program is executed by an interpreter. When the interpreter reaches certain points, called *join points*, in its execution, it invokes a *weaver*, passing to it an abstraction of its internal state (the *current join point*). Each advice contains a formula, called a *pointcut designator* (*pcd*), describing the join points in which it is interested, and a body representing the action to take at those points. It is the job of the weaver to distribute the join points from the interpreter, invoking each piece of advice that is interested in the current join point and executing its body with the same interpreter.

So far, this sounds like an instance of the Observer pattern [Gamma et al. 1995]. But there are several differences:

(1) When a piece of advice is run, its body may be evaluated before, after or instead of the expression that triggered it; this specification is part of the advice. In the last case, called an *around* advice, the advice body may call the primitive `proceed` to invoke the running of any other applicable pieces of advice and the base expression.
(2) The language of predicates is a temporal logic, with temporal operators such as `cflow` illustrated above. Hence the current join point may in general be an abstraction of the control stack.
(3) Each advice body is also interpreted by the same interpreter, so its execution may give rise to additional events and advice executions.
(4) Last, in the language of this article, as in the current implementation of AspectJ, the set of advice in each program is a global constant. This is in contrast with the Observer pattern, in which listeners register and deregister themselves dynamically.

This is of course a conceptual model and is intended only to motivate the semantics, not the implementation. However, this analysis highlights the major design decisions in any such language:

(1) The join-point model: when does the interpreter call the weaver, and what data does it expose?
(2) The pcd language: what is the language of predicates over join points? How is data from the join point communicated to the advice?
(3) The advice model: how does advice modify the execution of the program?

In this article, we explore one set of answers to these questions. Section 3 gives a brief description of the language and some examples. Section 4 presents the semantics. In Section 6 we describe related work, including some of our current research directions.

3. EXAMPLES

Our base language consists of a set of mutually-recursive first-order procedures with a call-by-value interpretation. The language is first-order: procedures are not expressed values. The language includes assignment in the usual call-by-value fashion: new storage is allocated for every binding of a formal parameter, and identifiers in expressions are automatically dereferenced.

Figure 1 shows a simple program in this language, using a Scheme-like syntax. We have two pieces of `around` advice that are triggered by a call to `fact`.[1] At each advice execution, `x` will be bound to the argument of `fact`. The program begins by calling `main`, which calls `fact`. The first advice body is triggered. Its body prints the `before1` message and then evaluates the `proceed` expression,

[1]As shown in these examples, the executable version of ASB includes types for arguments and results. The portion of ASB captured by our semantics is untyped.

```
(run
  '((procedure void main ()
      (write (fact 3)))
    (procedure int fact ((int n))
      (if (< n 1) 1
        (* n (fact (- n 1)))))
    (around
      (and
        (pcall int fact (int))
        (args (int x)))
      (let (((int y) 0))
        (write 'before1:)
        (write x) (newline)
        (set! y (proceed x))
        (write 'after1:)
        (write x) (write y) (newline)
        y))
    (around
      (and
        (pcall int fact (int))
        (args (int x)))
      (let (((int y) 0))
        (write 'before2:) (write x)
        (newline)
        (set! y (proceed x))
        (write 'after2:)
        (write x) (write y) (newline)
        y))))
```

prints:

```
before1: 3
before2: 3
before1: 2
before2: 2
before1: 1
before2: 1
before1: 0
before2: 0
after2: 0 1
after1: 0 1
after2: 1 1
after1: 1 1
after2: 2 2
after1: 2 2
after2: 3 6
after1: 3 6
6
```

Fig. 1. Example of around advice.

which proceeds with the rest of the execution. The execution continues by invoking the second advice, which behaves similarly, printing the `before2` message; its evaluation of the `proceed` expression executes the actual procedure `fact`, which calls `fact` recursively, which invokes the advice again. Eventually `fact` returns 1, which is returned as the value of the `proceed` expression. As each `proceed` expression returns, the remainder of each advice body is evaluated, printing the various `after` messages.

Each `around` advice has complete control of the computation. Further computation, including any other applicable advice, is undertaken only if the advice body calls `proceed`. For example, if the `(set! y (proceed x))` in the first advice were omitted, the output would be just

```
before1: 3
after1: 3 0
0
```

The value of `x` must be passed to the `proceed`. If the call to `proceed` in the second advice were changed to `(proceed (− x 1))`, then `fact` would be called with a

```
(run                                                    prints:
  '((procedure void main ()
      (write (+ (fact 6) (foo 4))))                     4 4
    (procedure int fact ((int n))                       4 3
      (if (= n 0) 1                                     4 2
        (* n (fact (- n 1)))))                          4 1
    (procedure int foo ((int n))                        4 0
      (fact n))                                         744
    (before (and
              (pcall int fact (int))
              (args (int y))
              (cflow
                (and
                  (pcall int foo (int))
                  (args (int x)))))
      (write x) (write y) (newline))))
```

Fig. 2. Binding variables with `cflow`.

different recursive argument. This design choice is intentional: changing the argument to `proceed` is a standard idiom in AspectJ. We explore some of the consequences of this choice in Section 5 below.

Our language also includes `before` and `after` advice, which are evaluated on entry to and on exit from the join point that triggers them. These forms of advice do not require an explicit call to `proceed` and are always executed for effect, not value.

The language of pointcut designators includes temporal operators as well. Figure 2 shows an advice that is triggered by a call to `fact` that occurs within the dynamic scope of a call to `foo`. This program prints $720+24 = 744$, but only the last four calls to `fact` (the ones during the call to `foo`) cause the advice to execute. The pointcut argument to `cflow` binds `x` to the argument of `foo`. Our language of pcd's includes several temporal operators.

The examples shown here are from the Aspect Sand Box (ASB) [Masuhara and Kiczales 2003]. ASB consists of a simple language of classes, methods, and objects, called BASE, and a number of separate languages for doing different styles of aspect-oriented programming. The intention is that the same base language be used with different weavers, representing different models of AOP. Here we deal with the weaver PA, which is intended to model the pointcuts-and-advice style of AspectJ. ASB, including BASE and PA, is implemented using an interpreter written in Scheme in the style of Friedman et al. [2001].

For the semantics, we have simplified our system still further by replacing the object-oriented language BASE with a simple first-order procedural language that we will call PROC, and by modifying the join point model accordingly. While much has been left out, the language of the semantics still models essential characteristics of AspectJ, including dynamic join points, pointcut designators, and `before`, `after`, and `around` advice. The semantics presented here reflects the design of ASB as of the summer of 2001.

4. SEMANTICS

We use a monadic semantics, using partial-function semantics whenever possible. In general, we use lower-case Roman letters to range over sets, and Greek letters to range over elements of partial orders.

Typical sets:

Sets

$v \in Val$	Expressed Values
$l \in Loc$	Locations
$s \in Sto$	Stores
$id \in Id$	Identifiers (program variables)
$pname, wname \in Pname$	procedure names

If X is a set, we will write X^* to denote the set of all finite sequences of elements of X; if x is a typical element of X (as v is a typical element of *Val* above), then x^* will be a typical element of X^*. If X is a set or a partial order, then *Optional*(X) denotes the set or partial order obtained by adjoining an element *None* to X. If X is a partial order, then the partial order on *Optional*(X) is defined by making *None* incomparable to any element of X.

If X is a partial order, then $(X)_\perp$ will denote the partial order consisting of X with a new element $\perp$ adjoined, with $\perp \leq x$ for any element $x \in X$. When X is a set, $(X)_\perp$ denotes the partial order obtained in a similar way, taking the discrete partial order on X ($x \leq y$ iff $x = y$).

4.1 Join Points

A join point is an abstraction of the control stack. The portion of the program state made visible to the advice consists of the following data:

Join points

$jp \in JP$ — Join Points

$jp ::= \langle\rangle \mid \langle k, pname, wname, v^*, jp\rangle$

$k ::= \texttt{pcall} \mid \texttt{pexecution} \mid \texttt{aexecution}$ — Join Point Kinds

A join point is either empty or consists of a kind, some data, and a previous join point. The join point $\langle \texttt{pcall}, f, g, v^*, jp\rangle$ represents a call to procedure f from procedure g, with a tuple of arguments v^*, and with previous join point *jp*. We will also use two more classes of join points: `pexecution` and `aexecution` join points represent execution of a procedure or advice body; in these join points the three data fields contain empty values. The presence of these join points allows us to use idioms like `(not (cflow (axecution)))` which suppresses execution of an advice inside another advice. The *wname* could be omitted at the expense of additional complication in the language of pointcut designators.

$$\begin{array}{lll} b & \in & Bnd = [Id \to Val\,] \ \text{Bindings} \\ r & \in & Bnd + \{Fail\} \end{array}$$

$$\begin{array}{rcc} b \vee r = b & Fail \wedge\ r = Fail & \neg b = Fail \\ Fail \vee\ r = r & b \wedge Fail = Fail & \neg Fail = [\,] \\ & b \wedge b' = b + b' & \end{array}$$

Fig. 3. Algebra of pcd results.

Our standard join-point constructors are *new-pcall-jp*, *new-aexecution-jp*, and *new-pexecution-jp*, which are defined by:

Join Point Constructors

$$\begin{aligned} \textit{new-pcall-jp}\ \textit{pname}\ \textit{wname}\ v^* &= \lambda\ jp\ .\ \langle \texttt{pcall},\ \textit{pname},\ \textit{wname},\ v^*,\ jp\rangle \\ \textit{new-pexecution-jp}\ \textit{pname} &= \lambda\ jp\ .\ \langle \texttt{pexecution},\ \textit{None},\ \textit{pname},\ \langle\rangle,\ jp\rangle \\ \textit{new-aexecution-jp} &= \lambda\ jp\ .\ \langle \texttt{aexecution},\ \textit{None},\ \textit{None},\ \langle\rangle,\ jp\rangle \end{aligned}$$

4.2 Pointcut Designators

A pointcut designator is a formula that specifies the set of join points to which a piece of advice is applicable. When applied to a join point, a pointcut designator either succeeds with a set of bindings, or fails.

The grammar of pcd's is given by:

Pointcut designators

$$\begin{array}{rl} pcd ::= & (\texttt{pcall}\ pname)\ |\ (\texttt{pwithin}\ pname) \\ | & (\texttt{aexecution})\ |\ (\texttt{pexecution}) \\ | & (\texttt{args}\ id_1 \ldots id_n) \\ | & (\texttt{and}\ pcd\ pcd)\ |\ (\texttt{or}\ pcd\ pcd)\ |\ (\texttt{not}\ pcd) \\ | & (\texttt{cflow}\ pcd) \\ | & (\texttt{cflowbelow}\ pcd)\ |\ (\texttt{cflowtop}\ pcd) \end{array}$$

The semantics of pcd's is given by a function *match-pcd* that takes a pcd and a join point and produces either a set of bindings (a finite partial map from identifiers to expressed values) or the singleton *Fail*.

Before defining *match-pcd*, we must define the operations on bindings and pcd results. We write [ ] for the empty set of bindings and + for concatenation of bindings. The behavior of repeated bindings under + is unspecified. The operations $\vee$, $\wedge$, and $\neg$ on the result of *match-pcd* are shown in Figure 3.

Note that both $\wedge$ and $\vee$ are short-cutting, so that $\vee$ prefers its first argument. The negation of any set of bindings (empty or not) is *Fail*, and the negation of *Fail* is an empty set of bindings.

$$\mathit{match\text{-}pcd}(\texttt{pcall}\ \mathit{pname})\ \langle k, \mathit{pname}', \mathit{wname}, v^*, \mathit{jp}\rangle$$
$$= \begin{cases} [\,] & \text{if } k = \texttt{pcall} \wedge \mathit{pname} = \mathit{pname}' \\ \mathit{Fail} & \text{otherwise} \end{cases}$$

$$\mathit{match\text{-}pcd}(\texttt{pwithin}\ \mathit{wname})\ \langle k, \mathit{pname}, \mathit{wname}', v^*, \mathit{jp}\rangle$$
$$= \begin{cases} [\,] & \mathit{wname} = \mathit{wname}' \\ \mathit{Fail} & \text{otherwise} \end{cases}$$

$$\mathit{match\text{-}pcd}(\texttt{pexecution})\ \langle k, \mathit{pname}, \mathit{wname}, v^*, \mathit{jp}\rangle$$
$$= \begin{cases} [\,] & \text{if } k = \texttt{pexecution} \\ \mathit{Fail} & \text{otherwise} \end{cases}$$

$$\mathit{match\text{-}pcd}(\texttt{aexecution})\ \langle k, \mathit{pname}, \mathit{wname}, v^*, \mathit{jp}\rangle$$
$$= \begin{cases} [\,] & \text{if } k = \texttt{aexecution} \\ \mathit{Fail} & \text{otherwise} \end{cases}$$

$$\mathit{match\text{-}pcd}(\texttt{args}\ \mathit{id}_1 \ldots \mathit{id}_n)\ \langle k, \mathit{pname}, \mathit{wname}, (v_1, \ldots, v_m), \mathit{jp}\rangle$$
$$= \begin{cases} [\mathit{id}_1 = v_1, \ldots, \mathit{id}_n = v_n] & \text{if } k = \texttt{pcall} \text{ and } n = m \\ \mathit{Fail} & \text{otherwise} \end{cases}$$

Fig. 4. *match-pcd*, part 1.

The definition of *match-pcd* proceeds by structural induction on its first argument. The pcd's fall into three groups.

(1) The first group, shown in Figure 4, does pattern matching on the top portion of the join point: `(pcall` *pname*`)` and `(pwithin` *pname*`)` check the target and within fields of the join point. `(args` $\mathit{id}_1 \ldots \mathit{id}_n$`)` succeeds if the argument list in the join point contains exactly n elements, and binds $\mathit{id}_1, \ldots, \mathit{id}_n$ to those values.
(2) The second group, shown in Figure 5, `(and` *pcd pcd*`)`, `(or` *pcd pcd*`)`, and `(not` *pcd*`)`, perform Boolean combinations on the results of their arguments, using the functions $\wedge$, $\vee$, and $\neg$ defined above.
(3) Last, we have the temporal operators `(cflow` *pcd*`)`, `(cflowbelow` *pcd*`)`, and `(cflowtop` *pcd*`)`, also shown in Figure 5. The pcd `(cflow` *pcd*`)` finds the latest (most recent) join point that satisfies *pcd*. `(cflowbelow` *pcd*`)` is just like `(cflow` *pcd*`)`, but it skips the current join point, beginning its search at the first preceding join point. `(cflowtop` *pcd*`)` is like `(cflow` *pcd*`)`, but it finds the earliest matching join point. These searches can be thought of as local loops within the overall structural induction.

4.3 The Execution Monad

We structure our semantics using a monad [Moggi 1991; Wadler 1992]. A monad consists of a datatype transformer T and two families of operations $\mathbf{return}_A$ and $\mathbf{seq}_{AB}$ with types

$$\mathbf{return}_A : A \to T(A)$$
$$\mathbf{seq}_{AB} : T(A) \to (A \to T(B)) \to T(B)$$

$$\begin{aligned}
&\mathit{match\text{-}pcd}(\texttt{and}\ \mathit{pcd}_1\ \mathit{pcd}_2)\,\mathit{jp} = \mathit{match\text{-}pcd}\,\mathit{pcd}_1\,\mathit{jp}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge \mathit{match\text{-}pcd}\,\mathit{pcd}_2\,\mathit{jp}\\
&\mathit{match\text{-}pcd}(\texttt{or}\ \mathit{pcd}_1\ \mathit{pcd}_2)\,\mathit{jp} = \mathit{match\text{-}pcd}\,\mathit{pcd}_1\,\mathit{jp}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \vee \mathit{match\text{-}pcd}\,\mathit{pcd}_2\,\mathit{jp}\\
&\mathit{match\text{-}pcd}(\texttt{not}\ \mathit{pcd})\,\mathit{jp} = \neg(\mathit{match\text{-}pcd}\,\mathit{pcd}\,\mathit{jp})
\end{aligned}$$

$$\begin{aligned}
&\mathit{match\text{-}pcd}(\texttt{cflow}\ \mathit{pcd})\ \langle\rangle = \mathit{Fail}\\
&\mathit{match\text{-}pcd}(\texttt{cflow}\ \mathit{pcd})\ \langle k, \mathit{pname}, \mathit{wname}, v^*, \mathit{jp}\rangle\\
&\quad = \mathit{match\text{-}pcd}\,\mathit{pcd}\ \langle k, \mathit{pname}, \mathit{wname}, v^*, \mathit{jp}\rangle\\
&\qquad \vee \mathit{match\text{-}pcd}(\texttt{cflow}\ \mathit{pcd})\,\mathit{jp}
\end{aligned}$$

$$\begin{aligned}
&\mathit{match\text{-}pcd}(\texttt{cflowbelow}\ \mathit{pcd})\ \langle\rangle = \mathit{Fail}\\
&\mathit{match\text{-}pcd}(\texttt{cflowbelow}\ \mathit{pcd})\ \langle k, \mathit{pname}, \mathit{wname}, v^*, \mathit{jp}\rangle\\
&\quad = \mathit{match\text{-}pcd}(\texttt{cflow}\ \mathit{pcd})\,\mathit{jp}
\end{aligned}$$

$$\begin{aligned}
&\mathit{match\text{-}pcd}(\texttt{cflowtop}\ \mathit{pcd})\ \langle\rangle = \mathit{Fail}\\
&\mathit{match\text{-}pcd}(\texttt{cflowtop}\ \mathit{pcd})\ \langle k, \mathit{pname}, \mathit{wname}, v^*, \mathit{jp}\rangle\\
&\quad = \ \mathit{match\text{-}pcd}(\texttt{cflowtop}\ \mathit{pcd})\,\mathit{jp}\\
&\qquad \vee \mathit{match\text{-}pcd}\,\mathit{pcd}\ \langle k, \mathit{pname}, \mathit{wname}, v^*, \mathit{jp}\rangle
\end{aligned}$$

Fig. 5. *match-pcd*, part 2.

The intention is that if A is a set of values, then $T(A)$ is a model of computations of that type, possibly including side effects. $\mathbf{return}_A$ maps an element $a \in A$ into the "constant computation" that always returns a; $\mathbf{seq}_{AB}\,c f$ is the computation which, when run, first runs c; if that computation returns the value a, then it runs the computation (fa).

For example, in the state monad, a computation of type A can be modelled by an element of $\mathit{Sto} \to (A \times \mathit{Sto})$, so we can define

$$\begin{aligned}
T(A) &= \mathit{Sto} \to (A \times \mathit{Sto})\\
\mathbf{return}_A &= \lambda s\,.\,(a, s)\\
\mathbf{seq}_A B &= \lambda c\,.\,\lambda f \cdot \lambda s\,.\,((\lambda\,(a, s')\,.\,((fa)s'))(cs))
\end{aligned}$$

These operations only define the "pure" operations; effects are defined by additional operators. For example, if the store consisted of a single integer (so $\mathit{Sto} = \mathit{Int}$), we might define additional operations

$$\begin{aligned}
\mathbf{get} &: T(\mathit{Int})\\
&= \lambda s\,.\,(s, s)\\
\mathbf{put} &: \mathit{Int} \to T(\mathit{Int})\\
&= \lambda n\,.\,\lambda s\,.\,(1, n)
\end{aligned}$$

Here **get** is an integer-valued computation that returns the contents of the single location in the store and **put** is an operation that takes an integer n and returns a computation that takes the store s and returns 1, leaving n in the store. Thus a computation that increments the store and returns the old value can be written as:

$$\mathbf{seq}(\mathbf{get}, (\lambda n.\,\mathbf{seq}(\mathbf{put}(n+1),\ \lambda d.\,\mathbf{return}\,n)).$$

We will omit the subscripts on **return** and **seq**, as we have above, whenever they are clear from context. The example also illustrates that the most common use of **seq** is in the form $\mathbf{seq}(E_1, \lambda\,v\,.\,E_2)$. For this we use the syntax $\mathbf{let}\,v \Leftarrow E_1\ \mathbf{in}\,E_2$, where v is bound in E_2. Using this notation, the preceding example might be written as:

$$\textbf{let}\, n \Leftarrow \textbf{get}\ \textbf{in}\,\textbf{let}\, d\ \Leftarrow \textbf{put}(n+1)\,\textbf{in}\,\textbf{return}\, n$$

let is similar to the Haskell **do** syntax [Peyton Jones et al. 1999]. We write

$$\textbf{let}\, v_1 \Leftarrow c_1; \ldots ; v_n \Leftarrow c_n\ \textbf{in}\, E$$

for the evident nested **let**. In this notation our example would appear as:

$$\textbf{let}\, n \Leftarrow \textbf{get};\ d\ \Leftarrow \textbf{put}(n+1)\,\textbf{in}\ \textbf{return}\, n$$

let and **seq** are interdefinable; we will define our advice monad using **let**.

These operations must obey a number of algebraic laws [Moggi 1991], which will guarantee, for example, that **seq** is associative, and that **let** behaves in the expected ways. Our monad will obey these laws, and we will not need to be concerned with the details.

There are standard ways of constructing monads for various effects. For our language, we introduce a monad:

$$T(A) = \mathit{JP} \to \mathit{Sto} \to (A \times \mathit{Sto})_\perp$$

This is a monad with three effects: a dynamically-scoped quantity of type *JP*, a store of type *Sto*, and nontermination. It says that a computation runs given a join point and a store, and either produces a value and a store, or else fails to terminate. The monad operations are defined as follows:

Monad operations

$$\begin{array}{l}
\textbf{return}\, v = \lambda \mathit{jp}\, s\,.\,(v, s)\\
\textbf{let}\, v \Leftarrow E_1\ \textbf{in}\, E_2\\
\quad = \lambda\ \mathit{jp}\, s\,.\,\textbf{case}\ (E_1\ \mathit{jp}\, s)\ \textbf{of}\\
\qquad\qquad\qquad\qquad \perp\ \Rightarrow\ \perp\\
\qquad\qquad\qquad\qquad (v, s') \Rightarrow ((E_2\, v)\ \mathit{jp}\, s')
\end{array}$$

The operation **return** returns the value, leaving the store unchanged and forgetting its join point. **let** $v \Leftarrow E_1$ **in** E_2 is a computation that, given a join point *jp* and a store s, does the following: first, it runs E_1 in the current join point and store. If this computation denotes $\perp$ (that is, if it fails or does not terminate), then the entire computation fails with denotation $\perp$. If it returns a value v and a store s', then the computation $(E_2\ v)$ is run using the new store s', but the original join point *jp*. These definitions ensure that *JP* has dynamic scope and that *Sto* is global.

For the store, we will have monadic operations **alloc**, which allocates a new location and returns it, **deref**, which takes a location and returns its value in the store, and **setref**, which takes a location and value and updates the location accordingly.

For join points we will have a single monadic operator **setjp**. **setjp** takes a function f from join points to join points and a map g from join points to computations. It returns a computation that, given a join point *jp* and a store

s, applies f to the current join point, passes the new join point $(f\,jp)$ to g, and runs the resulting computation $(g\ (f\,jp))$ on the new join point $(f\,jp)$ and the current store s. This is a somewhat convoluted operation, but it turns out to be well-matched to our application. Translating this into a λ-term, we get:

setjp

$$\begin{aligned}\textbf{setjp} &: (JP \to JP) \to (JP \to T(A)) \to T(A)\\ &= \lambda\, f\, g\, .\, \lambda\ jp\, s\, .\, (g\ (f\,jp))\ (f\,jp)\ s\end{aligned}$$

Since the codomain of $T(A)$ is the partial order $(A \times Sto)_\bot$, each $T(A)$ has an order defined by:

$$c \le c' \iff (\forall\ jp\, s)((c\, jp\, s) \le (c'\, jp\, s))$$

and from this order we can define additional domains with the usual pointwise order:

Domains

$\chi \in T(Val)$	Computations
$\pi \in Proc = Val^* \to T(Val)$	Procedures
$\alpha \in Adv = JP \to Proc \to Proc$	Advice
$\phi \in PE = Pname \to Proc$	Procedure Environments
$\gamma \in AE = Adv^*$	Advice Environments
$\rho \in Env = [Id \to Loc] \times WName \times Proceed$	Environments
$WName = Optional(Pname)$	`within` Info
$Proceed = Optional(Proc)$	`proceed` Info

A procedure takes a sequence of arguments and produces a computation. An advice takes a join point and a procedure, and produces a new procedure that either is the original procedure wrapped in the advice (if the advice is applicable at this join point) or else is the original procedure unchanged (if the advice is inapplicable). Procedures and advice do not require any environment arguments because they are always defined globally and are closed (mutually recursively) in the global procedure- and advice-environments.

The distinguished *WName* component of the environment will be used for tracking the name of the procedure (if any) in which the current program text resides. Similarly, the distinguished *Proceed* component will be used for the `proceed` operation, if it is defined. We write $\rho(\%\texttt{within})$, $\rho[\%\texttt{within} = \ldots]$, $\rho(\%\texttt{proceed})$, and $\rho[\%\texttt{proceed} = \ldots]$ to manipulate these components.

4.4 Expressions

We can now give the semantics of expressions. Figure 6 shows some key fragments.

$$
\begin{aligned}
&\mathcal{E}[\![e]\!] \in \mathit{Env} \to \mathit{PE} \to \mathit{AE} \to T(\mathit{Val})\\[1em]
&\mathcal{E}[\![(\mathit{pname}\ \ e_1\ \ \ldots\ \ e_n)]\!]\rho\phi\gamma\\
&\quad = \textbf{let}\, v_1 \Leftarrow \mathcal{E}[\![e_1]\!]\rho\phi\gamma\,;\ \ldots;\ v_n \Leftarrow \mathcal{E}[\![e_n]\!]\rho\phi\gamma\\
&\qquad \textbf{in}\ (\mathit{enter\text{-}join\text{-}point}\ \gamma\\
&\qquad\qquad (\mathit{new\text{-}pcall\text{-}jp}\ \mathit{pname}\ (\rho\ \texttt{\%within})\,(v_1,\ldots,v_n))\\
&\qquad\qquad (\phi\,(\mathit{pname}))\\
&\qquad\qquad (v_1,\ldots,v_n))\\[1em]
&\mathcal{E}[\![\texttt{(proceed}\ \ e_1\ \ \ldots\ \ e_n\texttt{)}]\!]\rho\phi\gamma\\
&\quad = \textbf{let}\, v_1 \Leftarrow \mathcal{E}[\![e_1]\!]\rho\phi\gamma\,;\ \ldots;\ v_n \Leftarrow \mathcal{E}[\![e_n]\!]\rho\phi\gamma\\
&\qquad \textbf{in}\ \rho\,(\texttt{\%proceed})(v_1,\ldots,v_n)
\end{aligned}
$$

Fig. 6. Semantics of expressions.

In a procedure call, first the arguments are evaluated in the usual call-by-value monadic way. Then, instead of directly calling the procedure, we use *enter-join-point* to create a new join point and enter it, invoking the weaver to apply any relevant advice. Contrast this with the `proceed` expression, which is like a procedure call, except that the special procedure `%proceed` is called, and no additional weaving takes place.

4.5 The Weaver and Advice

enter-join-point is the standard entry to a new join point. It takes a list of advice γ, a join-point transformer f, a procedure π, and a list of arguments v^*. It produces a computation that builds a new join point using function f, and then calls the weaver. The weaver then builds a new procedure, which is applied to v^* in the new join point. All this is accomplished using the monadic operation **setjp**:

enter-join-point

$$
\begin{aligned}
&\mathit{enter\text{-}join\text{-}point} : \mathit{AE} \to (\mathit{JP} \to \mathit{JP}) \to \mathit{Proc} \to \mathit{Proc}\\
&\quad = \lambda\,\gamma f \pi .\, \lambda v^* .\, \textbf{setjp}\, f\, (\lambda jp' .\, (\mathit{weave}\,\gamma\, jp'\, \pi)\, v^*)
\end{aligned}
$$

The weaver takes a list of advice, a join point, and a procedure. It returns a new procedure that consists of the original procedure wrapped in all of the advice that is applicable at the join point. To do this, the weaver attempts to apply each piece of advice in turn. If there is no advice left, then the effective procedure is just the original procedure π. Otherwise, it calls the first advice in the list, asking it to wrap its advice (if applicable) around the procedure that results from weaving the rest of the advice around the original procedure.

The Weaver

$$
\begin{aligned}
&\mathit{weave} : \mathit{AE} \to \mathit{JP} \to \mathit{Proc} \to \mathit{Proc}\\
&\quad = \lambda\gamma\ jp\pi\ .\, \textbf{case}\ \gamma\ \ \textbf{of}\\
&\qquad\qquad \langle\rangle \Rightarrow \pi\\
&\qquad\qquad \alpha :: \gamma' \Rightarrow \alpha\ jp\,(\mathit{weave}\,\gamma'\ jp\ \pi)
\end{aligned}
$$

Thus we have

$$(weave\,\langle \alpha_1, \ldots, \alpha_n\rangle\, jp\, \pi) = (\alpha_1\, jp\, (\alpha_2\, jp \cdots (\alpha_n\, jp\, \pi)\cdots)).$$

This brings us to the semantics of advice, shown in Figure 7. A piece of advice, like an expression, should take a procedure environment and an advice environment, and its meaning should be a procedure transformer like the α's above. Our fundamental model is `around` advice. If the advice does not apply in the current join point, then the procedure should be unchanged. If the advice does apply, then the advice body should be executed with the bindings derived from the pcd, and with `%proceed` set to the original procedure (which may be either the starting procedure or a procedure containing the rest of the woven advice). However, there are two subtleties: first, the body of the advice is to be executed in a new `aexecution` join point, so we use *enter-join-point* to build the new join point and invoke the weaver. This is potentially an infinite regress, so most advice pcd's will include an explicit `pcall` conjunct to avoid this problem. Second, in this case, the inner v^* is not used; the advice body can retrieve it using an `args` pcd.

`before` and `after` advice are similar. `%proceed` is not bound, and we use the monad operations to perform the sequencing.

The function $\mathcal{PCD}[\![-]\!]$ takes four arguments: a pcd, a join point, a function k from environments to computations (the "success continuation"), and a computation χ (the "failure computation"), and it produces a computation. It calls *match-pcd* to match the pcd against the join point. If *match-pcd* succeeds with a set of bindings, $\mathcal{PCD}$ creates an environment containing a fresh location for each binding, and invokes the success continuation on this environment, producing a new computation. Otherwise, it returns the failure computation.

Semantics of pcd's

$$\begin{aligned}
&\mathcal{PCD}[\![pcd]\!] : JP \to (Env \to T(Val)) \to T(Val) \to T(Val)\\
&\quad = \lambda\, jp\, k\, \chi\, .\, \mathbf{case}\,(match\text{-}pcd\, pcd\, jp)\ \mathbf{of}\\
&\qquad\qquad Fail \Rightarrow \chi\\
&\qquad\qquad [x_1 = v_1, \ldots, x_n = v_n] \Rightarrow\\
&\qquad\qquad\quad \mathbf{let}\, l_1 \Leftarrow \mathbf{alloc}(v_1); \ldots; l_n \Leftarrow \mathbf{alloc}(v_n)\\
&\qquad\qquad\quad \mathbf{in}\, k([x_1 = l_1, \ldots, x_n = l_n])
\end{aligned}$$

4.6 Procedures and Programs

Finally, we give the semantics of procedure declarations and whole programs. The meaning of a procedure declaration in a procedure and advice environment is a procedure environment containing a single binding. In this binding, the name of the procedure is bound to a procedure that accepts some arguments and enters a `pexecution` join point, weaving any applicable advice from the advice environment along the way. When the advice is accounted for, the arguments are stored in new locations, and the procedure body is executed in an environment in which the formal parameters are bound to the new locations, as is typical in call-by-value languages.

```
$\mathcal{A}[\![$((around $pcd$) $e$)$]\!]\phi\gamma : JP \to Proc \to Proc$
  $= \lambda jp\, \pi\, v^*.$
      $\mathcal{PCD}[\![pcd]\!]jp$
        $(\lambda\rho.enter\text{-}join\text{-}point\ \gamma$
           $new\text{-}aexecution\text{-}jp$
           $(\lambda v^*. \mathcal{E}[\![e]\!](\rho[$%within $= None,$
                          %proceed $= \pi])\phi\gamma)$
           $\langle\rangle)$
        $(\pi\ v^*)$

$\mathcal{A}[\![$((before $pcd$) $e$)$]\!]\phi\gamma : JP \to Proc \to Proc$
  $= \lambda jp\, \pi\, v^*.$
      $\mathcal{PCD}[\![pcd]\!]jp$
        $(\lambda\rho.enter\text{-}join\text{-}point\ \gamma$
           $new\text{-}aexecution\text{-}jp$
           $(\lambda v^*.$ let
               $v_1 \Leftarrow \mathcal{E}[\![e]\!](\rho[$%within $= None,$
                              %proceed $= None])\phi\gamma;$
               $v_2 \Leftarrow (\pi\ v^*)$
               in $v_2)$
           $\langle\rangle)$
        $(\pi\ v^*)$

$\mathcal{A}[\![$((after $pcd$) $e$)$]\!]\phi\gamma : JP \to Proc \to Proc$
  $= \lambda jp\, \pi\, v^*.$
      $\mathcal{PCD}[\![pcd]\!]jp$
        $(\lambda\rho.enter\text{-}join\text{-}point\ \gamma$
           $new\text{-}aexecution\text{-}jp$
           $(\lambda v^*.$ let
               $v_1 \Leftarrow (\pi\ v^*);$
               $v_2 \Leftarrow \mathcal{E}[\![e]\!](\rho[$%within $= None,$
                              %proceed $= None])\phi\gamma$
               in $v_1)$
           $\langle\rangle)$
        $(\pi\ v^*)$
```

Fig. 7. Semantics of advice.

Semantics of procedure declarations

```
$\mathcal{P}[\![$(procedure $pname$ $(x_1 \ldots x_n)$ $e$)$]\!] : PE \to AE \to PE$
  $= \lambda\,\phi\,\gamma\,.[pname =$
          $\lambda v^*\,.(enter\text{-}join\text{-}point\ \gamma$
               $(new\text{-}pexecution\text{-}jp\ pname)$
               $(\lambda w\,.$ let $l_1 \Leftarrow$ alloc$(w{\downarrow}1)\,;$
                          $\vdots$
                          $l_n \Leftarrow$ alloc$(w{\downarrow}n)$
                   in $(\mathcal{E}[\![e]\!][x_1 = l_1,\ \ldots,x_n = l_n,$
                              %within $= pname,$
                              %proceed $= None]\ \phi\ \gamma))$
               $v^*)]$
```

```
((procedure void main () (fib 19))
 (procedure int fib ((int x)) (fib x))
 (around
   (and
     (pcall int fib (int))
     (args (int n)))
   (if (< n 2) 1
     (let (((int n1) (proceed (- n 1)))
           ((int n2) (proceed (- n 2))))
       (+ n1 n2))))))
```

Fig. 8. Convoluted fibonacci function.

We have formulated the semantics of procedures and advice as being closed in a given procedure environment and advice environment. A program is a mutually recursive set of procedures and advice, so its semantics is given by the fixed point over these functions. We take the fixed point and then apply the procedure `main` to no arguments.

Semantics of programs

$$\mathcal{PGM}[\![(proc_1 \ldots proc_n\; adv_1 \ldots adv_m)]\!] : T(Val)$$
$$= run\big(fix\big(\lambda(\phi, \gamma)\,.\,\big(\textstyle\sum_{i=1}^{n}(\mathcal{P}[\![proc_i]\!]\phi\gamma), \langle\mathcal{A}[\![adv_j]\!]\phi\gamma\rangle_{j=1}^{m}\big)\big)\big)$$

$$run(\phi, \gamma) = \mathcal{E}[\![(\texttt{main})]\!][\,]\phi\gamma$$

Here the notation $\langle\cdots\rangle_{j=1}^{m}$ denotes a sequence of length m, and the notation $\sum_{i=1}^{n}$ denotes the concatenation operator on bindings, discussed on page 896.

This completes the semantics of the core language.

5. LESSONS LEARNED

One of the standard claims for formal methods is that they shed light on the system being studied. We therefore discuss some of what we learned in the process of formalizing this language.

5.1 Some Pathological Programs

Figure 8 shows an unusual program. To understand this code, consider the call `(fib 19)` in `main`. This call matches the pcd of the advice, so the advice body is executed with `n` bound to 19. The advice body then calls `proceed` on the value 18. This causes the body of `fib` to be executed with `x` bound to 18. However, the body of `fib` is another call to `fib`, so the advice is executed again with `n` bound to 18. This process continues recursively, and the program correctly calculates the fibonacci function.

One could transform any recursive procedure similarly, moving the body of the procedure into the advice. This process may be understood as defining the procedure as the trivial infinitely-looping procedure, with the advice helping the procedure by managing the recursion, including terminating when possible.

```
((procedure int main () (times 3 5))
 (procedure int times ((int x0) (int y0)) (loop x0 0))
 (procedure int loop ((int x) (int acc)) (loop x acc))
 (around
   (and
     (pcall int loop (int int))
     (args (int x) (int acc))
     (cflow (and
              (pcall int times (int int))
              (args (int x0) (int y0)))))
   (if (= 0 x)
     acc
     (proceed (- x 1) (+ acc y0)))))
```

Fig. 9. Convoluted multiplication function.

Figure 9 shows a still more convoluted program. This program is a variation on the standard implementation of multiplication as repeated addition, which might be written in a conventional imperative language as:

```
proc (int x0, int y0) {
  x = x0; acc = 0;
  while x != 0 {
    x = x-1;
    acc = acc + y0;
    };
  return acc; }.
```

The inner loop is realized as the recursive procedure `loop`, and the body of the loop is moved into the advice, as before. The new wrinkle here is the variable `y0`. Although `y0` is not an argument to the inner loop, it is available in the join point and is bound by the `cflow`. The `cflow` effectively allows access to dynamically-bound variables.

Of course, neither of these programs represents recommended AOP programming style, but they illustrate the power of advice. Only the program in Figure 9 begins to illustrate the power of pointcuts to refer to join points in ways that crosscut the program structure, which is the more novel and important part of AOP.

5.2 Arguments to `proceed`

In our language, `proceed` takes as its arguments the arguments to the underlying procedure. This is a reasonable choice in our language, when advice is typically wrapped around procedure calls. However, it leads to an intriguing discontinuity: if there is only one applicable advice at a procedure call, then `proceed` is bound to the execution of the underlying procedure body, and the argument to `proceed` is passed to the procedure, as in the preceding examples. If, however, there are two pieces of applicable `around` advice, then `proceed` in the first advice is bound not to the execution of the procedure, but to the execution of the second advice, as in Figure 1. But the variables of an advice body are bound by its pointcut descriptor, not from any arguments that the advice might be passed. To see this, note in the semantics of advice that the outermost v^* does not occur free within the success branch of $\mathcal{PCD}[\![-]\!]$. Hence the argument to the proceed is ignored. For example, in Figure 1, if the first advice were changed to

```
(around
  ...
  (let (((int y) 0))
    ...
    (set! y (proceed 23))
    ...
    y))
```

the program would behave just as it did before.

In addition to this semantic discontinuity, this means that multiple around advice interacts in a somewhat surprising way: the outermost advice gets to decide whether or not the procedure (along with any inner advice) gets executed, but the innermost advice gets to decide what parameters are passed to the procedure body (discarding any contributions from outer advice). Among other things, this makes the notion of "precedence" problematical for around advice in this design.

AspectJ uses a different design for the arguments to proceed. In AspectJ, the proceed form takes as arguments the values exposed by the around's pointcut. This is convenient, since in AspectJ, pointcut designators have explicitly declared bound variables, for example:

```
(int i) : call(int Foo.m(int)) && args(i)
```

whereas the arguments to a pointcut designator in ASB must be reconstructed from the pcd.

However, not every argument exposed in a pointcut may be meaningfully changed. For example, in the pointcut

```
(and
  (pcall int fact (int))
  (args (int y))
  (cflow
    (and
      (pcall int foo (int))
      (args (int x)))))
```

it is not clear what it might mean to change the value of x, since that refers to a procedure call that has already taken place.

In response to these issues, the design of ASB has evolved since the 2001 snapshot on which this semantics was based. For example, one version of ASB, in Spring, 2002, separated the variables of a pcd into those that may be meaningfully changed and those that may not. However, this design does not help with the problem of passing information to an inner around advice. One could imagine writing the arguments back into the join point. But then the data in the join point would no longer be causally connected to the underlying program. Worse yet, the inner advice was selected for execution based on the original join point; with the modified join point, the inner advice might no longer even be applicable. The version of ASB as of October, 2002 [Dutchyn et al. 2002] has `proceed` evaluate its parameters, but the values of the parameters are ignored entirely. This design precludes the tricks of Figures 8 and 9.

5.3 Type Safety for `proceed`

Although the language captured by our semantics is untyped, we have considered some typing issues.

In general, it appears difficult or impossible to deduce a type for the procedure bound to `proceed`. In our language, we would have to deduce a type for `proceed` in the presence of disjunction and negation in the pointcut descriptors. In AspectJ, this problem is even more difficult because pointcut descriptors may be abstract.

AspectJ deals with this probem by imputing the return type of a `proceed` to be the same as the return type of the advice. This is unsound in general, since the value returned by the `proceed` depends on the code under the join point, not on the code in the advice.

Even if we could assign a sound type to `proceed`, the architecture of the system requires that the result type of `proceed` be invariant, that is, neither covariant nor contravariant, in the presence of subtyping. To demonstrate this, imagine we had a program with a procedure `f : () -> A` and an advice that we might write in our Scheme-like syntax, but with types, as

```
(B around
  (pcall A f ())
  (proceed))
```

where `A` is the declared result type of `f` and `B` is the declared result type of the advice. Such an advice, which merely calls the underlying procedure, should be typable. But if this is to be well-typed, then the value of `(proceed)` (known to be of type `A`), must be of type `B`. So we must have `A < B`.

On the other hand, consider the advice

```
(B around
  (pcall A f ())
  (new B))
```

If this advice is to be well-typed, then the value of the advice body (known to be of type `B`), must be of type `A` in order to be substitutable for the original call to `f`. So `B < A`.

This invariance appears to be part of the essential semantics of `proceed`, independent of how the arguments are to be treated.

6. RELATED WORK

Aspect-oriented programming is presented in Kiczales et al. [1997], which shows how several elements of prior work, including reflection [Smith 1984], metaobject protocols [Kiczales and des Rivieres 1991], subject-oriented programming [Harrison and Ossher 1993], adaptive programming [Lieberherr

1996], and composition filters [Aksit et al. 1994] all enable better control over modularization of crosscutting concerns. A variety of models of AOP are presented in Elrad et al. [2001]. AspectJ [Kiczales et al. 2001] is a Java-based language explicitly driven by the principles of AOP.

Flavors [Weinreb and Moon 1981; Cannon 1982], New Flavors [Moon 1986], CommonLoops [Bobrow et al. 1986] and CLOS [Steele 1990] all support `before`, `after`, and `around` methods.

De Meuter [1997] essays an explanation of AOP in terms of monads. He introduces aspects by modifying the **seq** function to take additional arguments. This has the effect of making advice part of the metalanguage rather than of the language itself. Also it is not clear when such a modified **seq** function would satisfy the monad laws. In our system, **seq** is fixed and satisfies the monad laws; different join point models can easily be accomodated.

Andrews [2001] presents a semantics for AOP programs based on a CSP formalism, using CSP synchronization sets as join points. His is an imperative language with first-order procedures, like ours, but it does not allow procedures to be recursive. His language includes `before`, `after`, and `around` advice, but his pcd's contain neither Boolean nor temporal operators.

Douence et al. [2001] present an event-based theory of AOP. They present a domain-specific language for defining "crosscuts" analagous to our pointcuts. Their language is very powerful, but its semantics is given by a rewriting semantics. Thus the semantics depends on details of the rewriting algorithm. We believe that our definition of *match-pcd* represents a significant improvement.

Lämmel [2002] presents static and dynamic operational semantics for a small OO language with a method-call interception facility. His system allows dynamic per-object attachment of advice, but does not treat `around` advice, with the attendant possibility of never executing the underlying method, nor does he deal with `cflow`-like pointcuts.

Masuhara et al. [2002] show how to compile a language like PA(BASE) into Scheme by partially evaluating the interpreter.

Tucker and Krishnamurthi [2003] present an operational semantics for an extension of pointcuts and advice to higher-order languages. Walker et al. [2003] present a core calculus for first-class aspects, which they claim is applicable to multiple languages, including both higher-order and object-oriented languages.

Masuhara and Kiczales [2003] present several other models of AOP and weaving in ASB, including static join points, Demeter [Lieberherr 1996], and Hyper/J [Ossher and Tarr 2000], and present a general model in which such systems can be described.

Jagadeesan et al. [2003] deal with the correctness of static weaving: that is, translating a program with aspects into a program in the same language without aspects. They present an operational semantics for an untyped base language with classes and objects. They then extend the language with pointcuts and advice. They give a direct operational semantics for the language with advice, and a translation from the language with advice to an equivalent language without advice, and show that the translation preserves the operational semantics. However, their translation does not deal with `cflow`-like pointcuts.

REFERENCES

AKSIT, M., WAKITA, K., BOSCH, J., BERGMANS, L., AND YONEZAWA, A. 1994. Abstracting object interactions using composition filters. In *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, R. Guerraoui, O. Nierstrasz, and M. Riveill, Eds. Lecture Notes in Computer Science, vol. 791. Springer-Verlag, Berlin, Heidelberg, and New York, 152–184.

ANDREWS, J. H. 2001. Process-algebraic foundations of aspect-oriented programming. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*. Lecture Notes in Computer Science, vol. 2192. Springer-Verlag, Berlin, Heidelberg, and New York, 187–209.

BOBROW, D. G., KAHN, K., KICZALES, G., MASINTER, L., STEFIK, M., AND ZDYBEL, F. 1986. CommonLoops: merging Common Lisp and object-oriented programming. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, New York, 17–29.

CANNON, H. I. 1982. *Flavors: A Non-Hierarchical Approach to Object-Oriented Programming*. Symbolics, Inc., Cambridge, Mass.

DEMEUTER, W. 1997. Monads as a theoretical foundation for AOP. In *International Workshop on Aspect-Oriented Programming at ECOOP*. 25.

DOUENCE, R., MOTELET, O., AND SUDHOLT, M. 2001. A formal definition of crosscuts. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*. Lecture Notes in Computer Science, vol. 2192. Springer-Verlag, Berlin, Heidelberg, and New York, 170–186.

DUTCHYN, C., KICZALES, G., AND MASUHARA, H. 2002. Aspect sand box. http://www.cs.ubc.ca/labs/spl/projects/asb.html. web site.

ELRAD, T., FILMAN, R. E., AND BADER, A., Eds. 2001. *Comm. ACM. 44*, 10. ACM. special issue on Aspect-Oriented Programming.

FRIEDMAN, D. P., WAND, M., AND HAYNES, C. T. 2001. *Essentials of Programming Languages*, Second ed. MIT Press, Cambridge, MA.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts.

HARRISON, W. AND OSSHER, H. 1993. Subject-oriented programming (A critique of pure objects). In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, A. Paepcke, Ed. ACM Press, New York, 411–428.

JAGADEESAN, R., JEFFREY, A., AND RIELY, J. 2003. A calculus of untyped aspect-oriented programs. In *Proceedings of the European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, and New York.

KICZALES, G. AND DES RIVIERES, J. 1991. *The art of the metaobject protocol*. MIT Press, Cambridge, MA, USA.

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSEN, M., PALM, J., AND GRISWOLD, W. G. 2001. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 2072. Springer-Verlag, Berlin, Heidelberg, and New York, 327–353.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Vol. 1241. Springer-Verlag, Berlin, Heidelberg, and New York, 220–242.

LÄMMEL, R. 2002. A semantical approach to method-call interception. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, G. Kiczales, Ed. ACM Press, Enschede, The Netherlands.

LIEBERHERR, K. J. 1996. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, Mass.

MASUHARA, H. AND KICZALES, G. 2003. Crosscutting in aspect-oriented mechanisms. In *Proceedings of the European Conference on Object-Oriented Programming*. ACM Press, New York.

MASUHARA, K., KICZALES, G., AND DUTCHYN, C. 2002. Compilation semantics of aspect-oriented programs. In *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages Workshop at*

AOSD 2002, G. T. Leavens and R. Cytron, Eds. Tech. Rep. 02-06. Department of Computer Science, Iowa State University, 17–26.

Moggi, E. 1991. Notions of computation and monads. *Information and Computation 93*, 1, 55–92.

Moon, D. A. 1986. Object-oriented programming with Flavors. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, N. Meyrowitz, Ed. ACM Press, New York, NY, 1–8.

Ossher, H. and Tarr, P. 2000. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of the 22nd International Conference on Software Engineering, June 4–11, Limerick, Ireland*. ACM, New York, 734–737.

Peyton Jones, S., Ed. 1999. Haskell 98—a non-strict, purely functional language. Available from http://www.haskell.org/onlinereport.

Smith, B. C. 1984. Reflection and semantics in Lisp. In *Conference Record of the 11th ACM Symposium on Principles of Programming Languages*. ACM, New York, 23–35.

Steele, G. L. 1990. *Common Lisp: the Language*, Second ed. Digital Press, Burlington, MA.

Tucker, D. B. and Krishnamurthi, S. 2003. Pointcuts and advice in higher-order languages. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, M. Akşit, Ed. ACM Press, New York, 158–167.

Wadler, P. 1992. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, Albequerque, New Mexico, 1–14.

Walker, D., Zdancewic, S., and Ligatti, J. 2003. A theory of aspects. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. ACM, Uppsala, Sweden.

Weinreb, D. and Moon, D. A. 1981. Flavors: Message passing in the LISP machine. A. I. Memo 602, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts.

Received July 2002; accepted April 2003