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* to incrementally modify the behavior of a program. An advice declaration specifies an action to be taken whenever some condition arises during the execution of the program. The condition is specified by a formula called a *pointcut designator* or *pcd*. The events during execution at which advice may be triggered are called *join points*. In this model of aspect-oriented programming, join points are dynamic in that they refer to events during the execution of the program.

We give a denotational semantics for a minilanguage that embodies the key features of dynamic join points, pointcut designators, 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: []:

General Terms:

Additional Key Words and Phrases:

1. INTRODUCTION

A characteristic of aspect-oriented programming, as embodied in AspectJ [Kiczales et al. 2001], is the use of *advice* to incrementally modify the behavior of a program. An advice declaration specifies an action to be taken whenever some condition arises

Earlier versions of this paper 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 US National Science Foundation under grants CCR-9804115 and CCR-0097740, by the Natural Sciences Research Council (NSERC) of Canada, Sierra Systems Group and Xerox Canada Limited.
Authors' addresses: Mitchell Wand, College of Computer Science, Northeastern University, 360 Huntington Avenue, Boston, MA 02115, USA; email: m.wand@neu.edu. Gregor Kiczales and Christopher Dutchyn: Department of Computer Science, University of British Columbia, 201-2366 Main Mall, Vancouver, BC V6T 1Z4, Canada; email: {gregor,cdutchyn}@cs.ubc.ca
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0164-0925/20YY/0500-0001 $5.00

during the execution of the program. The events at which advice may be triggered are called *join points*. In this model of aspect-oriented programming (AOP), join points are *dynamic* in that they refer to events during execution. The process of executing the relevant advice at each join point is called *weaving*.

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

```
(and (pcalls f) (pwithin g) (cflow (pcalls 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 paper 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 Sandbox (ASB) project. The goal is of ASB 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. We hope to extend this work to other AOP models, including static join points, Demeter [Lieberherr 1996], and Hyper/J [Ossher and Tarr 2000], and to both interpreter-like and compiler-like implementation models.

For more motivation for AOP, see Kiczales et al. [1997] or the articles in [CACM 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 predicate, 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 demultiplex 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) First, 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) Second, 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 paper, 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 de-register 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 paper, we explore one set of answers to these questions. Section 3 gives brief description of the language and some examples. Section 4 presents the semantics. In section 6 we describe some related work, and in section 7 we discuss 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 the syntax of ASB. 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 in turn calls `fact`. The first advice body is triggered. Its body prints the `before1` message and then evaluates the `proceed` expression, 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

[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
        (pcalls 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
        (pcalls 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

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 `proceed` in the first advice were omitted, the output would be just

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

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

prints:

```
4 4
4 3
4 2
4 1
4 0
744
```

Fig. 2. Binding variables with `cflow`

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 "wrong" 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 of `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 of `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. For example, `cflowtop` finds the oldest contained join point that satisfies its argument. Our semantics includes a formal model that explains this behavior.

The examples shown here are from the Aspect Sandbox (ASB). ASB consists of a base language, called BASE, and a separate language of advice and weaving, called AJD. The language BASE is a simple language of procedures, classes, and objects. The intention is that the same base language be used with different weavers, representing different models of AOP; AJD is intended to capture the AspectJ dynamic join point style of AOP. The relation between AJD and BASE is intended to model the relationship between AspectJ and Java. We implemented the base language

and AJD using an interpreter in Scheme in the style of Friedman et al. [2001].

For the semantics, we have simplified BASE and AJD still further by removing types, classes, and objects from the language and by slightly simplifying the join point model; the details are listed in the appendix. 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.

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

4.1 Join Points

We begin with the definition of join points. We use the term *join point* to refer both to the events during the execution of the program at which advice may run and to the portion of the program state that may be visible to the advice. The portion of the program state made visible to the advice consists of the following data:

Join points

jp	$\in$	*JP*	Join Points
jp	$::=$	$\langle\rangle \mid \langle k, \mathit{pname}, \mathit{wname}, v^*, \mathit{jp}\rangle$	
k	$::=$	`pcall` $\mid$ `pexecution` $\mid$ `aexecution`	Join Point Kinds

A join point is an abstraction of the control stack. It is either empty or consists of a kind, some data, and a previous join point. The join point $\langle$`pcall`, f, g, v^*, $jp\rangle$ represents a call to procedure f from procedure g, with arguments v^*, and with previous join point jp. `pexecution` and `aexecution` join points represent execution of a procedure or advice body; in these join points the three data fields contain

empty values.

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

$$
\begin{aligned}
\mathit{pcd} &::= \texttt{(pcalls } \mathit{pname}\texttt{)} \mid \texttt{(pwithin } \mathit{pname}\texttt{)} \\
&::= \texttt{(args } \mathit{id}_1 \ \ldots \ \mathit{id}_n\texttt{)} \\
&::= \texttt{(and } \mathit{pcd}\ \mathit{pcd}\texttt{)} \mid \texttt{(or } \mathit{pcd}\ \mathit{pcd}\texttt{)} \mid \texttt{(not } \mathit{pcd}\texttt{)} \\
&::= \texttt{(cflow } \mathit{pcd}\texttt{)} \\
&::= \texttt{(cflowbelow } \mathit{pcd}\texttt{)} \mid \texttt{(cflowtop } \mathit{pcd}\texttt{)}
\end{aligned}
$$

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 defined by

Algebra of pcd results

$$
\begin{aligned}
b &\in Bnd = [Id \to Val] && \text{Bindings} \\
r &\in Optional\ (Bnd) = Bnd + \{Fail\}
\end{aligned}
$$

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

Note that both $\wedge$ and $\vee$ are short-cutting, so that $\vee$ prefers its first argument.

The definition of *match-pcd* is shown in figure 3. *match-pcd* proceeds by structural induction on its first argument. The pcd's fall into three groups.

(1) The first group does pattern matching on the top portion of the join point: `(pcalls` *pname*`)` and `(pwithin` *pname*`)` check the target and within fields of the join point. `(args` id_1 ... id_n`)` succeeds if the argument list in the join point contains exactly n elements, and binds $id_1, \ldots, id_n$ to those values. In full AJD, the `args` pcd includes dynamic type checks as well.

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

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

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

$$\textit{match-pcd}\ (\texttt{and}\ \textit{pcd}_1\ \textit{pcd}_2)\ \textit{jp} = \textit{match-pcd}\ \textit{pcd}_1\ \textit{jp} \wedge \textit{match-pcd}\ \textit{pcd}_2\ \textit{jp}$$

$$\textit{match-pcd}\ (\texttt{or}\ \textit{pcd}_1\ \textit{pcd}_2)\ \textit{jp} = \textit{match-pcd}\ \textit{pcd}_1\ \textit{jp} \vee \textit{match-pcd}\ \textit{pcd}_2\ \textit{jp}$$

$$\textit{match-pcd}\ (\texttt{not}\ \textit{pcd})\ \textit{jp} = \neg(\textit{match-pcd}\ \textit{pcd}\ \textit{jp})$$

$$\textit{match-pcd}\ (\texttt{cflow}\ \textit{pcd})\ \langle\rangle = \textit{Fail}$$

$$\textit{match-pcd}\ (\texttt{cflow}\ \textit{pcd})\ \langle k, \textit{pname}, \textit{wname}, v^*, \textit{jp}\rangle = \textit{match-pcd}\ \textit{pcd}\ \langle k, \textit{pname}, \textit{wname}, v^*, \textit{jp}\rangle \vee \textit{match-pcd}\ (\texttt{cflow}\ \textit{pcd})\ \textit{jp}$$

$$\textit{match-pcd}\ (\texttt{cflowbelow}\ \textit{pcd})\ \langle\rangle = \textit{Fail}$$

$$\textit{match-pcd}\ (\texttt{cflowbelow}\ \textit{pcd})\ \langle k, \textit{pname}, \textit{wname}, v^*, \textit{jp}\rangle = \textit{match-pcd}\ (\texttt{cflow}\ \textit{pcd})\ \textit{jp}$$

$$\textit{match-pcd}\ (\texttt{cflowtop}\ \textit{pcd})\ \langle\rangle = \textit{Fail}$$

$$\textit{match-pcd}\ (\texttt{cflowtop}\ \textit{pcd})\ \langle k, \textit{pname}, \textit{wname}, v^*, \textit{jp}\rangle = \textit{match-pcd}\ (\texttt{cflowtop}\ \textit{pcd})\ \textit{jp} \vee \textit{match-pcd}\ \textit{pcd}\ \langle k, \textit{pname}, \textit{wname}, v^*, \textit{jp}\rangle$$

Fig. 3. *match-pcd*

(2) The second group, `(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*`)`. 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 local loops within the overall structural induction.

4.3 The Execution Monad

To package the execution, we introduce a monad:

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

This is a monad with three effects: a dynamically-scoped quantity of type *JP*, a store of type *Sto*, and non-termination. 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 ensure that *JP* has dynamic scope and that *Sto* is global:

Monad operations

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

We write

$$\textbf{let}\ v_1 \Leftarrow \mu_1; \ldots; v_n \Leftarrow \mu_n\ \textbf{in}\ E$$

for the evident nested **let**.

We will have the usual monadic operations on the store; 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 applies f to the current join point, passes the new join point to g, and runs the resulting computation in the new join point and current store:

setjp

$$\begin{array}{l}\textbf{setjp} : (JP \to JP) \to (JP \to T(A)) \to T(A) \\ \quad = \lambda\ f\ \ g\ .\ \lambda\ jp\ s\ .\ (g\ \ (f\ \ jp))\ \ (f\ \ jp)\ \ s\end{array}$$

The *lift* operation induces an order on $T(A)$ for any A. We will use the following domains based on this 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 is either 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 ρ(`%within`), ρ[`%within` = ...], ρ(`%proceed`), and ρ[`%proceed` = ...] to manipulate these components.

4.4 Expressions

We can now give the semantics of expressions. Figure 4 shows some key fragments.

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. The function *new-pcall-jp* : $Pname \to WName \to Val^* \to JP \to JP$ builds a new procedure-call join point following the grammar in section 4.1.

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 builder f, a procedure π, and a list of arguments v^*. It produces a computation that builds a new join point using function f, calls the weaver to wrap all the advice in γ around procedure π, and then applies the resulting procedure to v^*.

$$\mathcal{E}[\![e]\!] \in \mathit{Env} \to \mathit{PE} \to \mathit{AE} \to T(\mathit{Val})$$

$$\begin{array}{l}\mathcal{E}[\![(\mathit{pname}\ \ e_1\ \ \ldots\ \ e_n)]\!]\rho\phi\gamma \\ \quad = \ \mathbf{let}\ v_1 \Leftarrow \mathcal{E}[\![e_1]\!]\rho\phi\gamma\,;\ \ldots\,;\ v_n \Leftarrow \mathcal{E}[\![e_n]\!]\rho\phi\gamma \\ \qquad \mathbf{in}\ (\mathit{enter\text{-}join\text{-}point}\ \ \gamma \\ \qquad\qquad (\mathit{new\text{-}pcall\text{-}jp}\ \ \mathit{pname}\ \ (\rho\ \texttt{\%within})\ \ (v_1, \ldots, v_n)) \\ \qquad\qquad (\phi\ (\mathit{pname})) \\ \qquad\qquad (v_1, \ldots, v_n))\end{array}$$

$$\begin{array}{l}\mathcal{E}[\![(\texttt{proceed}\ \ e_1\ \ \ldots\ \ e_n)]\!]\rho\phi\gamma \\ \quad = \mathbf{let}\ v_1 \Leftarrow \mathcal{E}[\![e_1]\!]\rho\phi\gamma\,;\ \ldots\,;\ v_n \Leftarrow \mathcal{E}[\![e_n]\!]\rho\phi\gamma \\ \qquad \mathbf{in}\ \rho\,(\texttt{\%proceed})\,(v_1, \ldots, v_n)\end{array}$$

Fig. 4. Semantics of expressions

enter-join-point

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

The weaver is the heart of the system. It 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.

So we want

$$(\mathit{weave}\ \ \langle\alpha_1, \ldots, \alpha_n\rangle\ \ \mathit{jp}\ \ \pi) = (\alpha_1\ \ \mathit{jp}\ \ (\alpha_2\ \ \mathit{jp}\ \ \ldots\ \ (\alpha_n\ \ \mathit{jp}\ \ \pi)\ldots))$$

This becomes a straightforward bit of functional programming, shown in figure 5

This brings us to the semantics of advice, shown in figure 6. A piece of advice, like an expression, should take a procedure environment and an advice environment, and its meaning should be a procedure transformer. 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,

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

Fig. 5. The weaver

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 `pcalls` 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

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

4.6 Procedures and Programs

Finally, we give the semantics of procedures and whole programs. The meaning of a procedure in a procedure and advice environment is a small procedure environment. In this environment, the name of the procedure is bound to a procedure that accepts some arguments and enters a `pexecution` join point, possibly weaving some advice. 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

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

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

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

Fig. 6. Semantics of advice

are bound to the new locations.

Semantics of procedure declarations

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

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

$$
\begin{array}{l}
\mathcal{PGM}[\![(\ \mathit{proc}_1\ \ \ldots\ \ \mathit{proc}_n\ \ \mathit{adv}_1 \ldots\ \ \mathit{adv}_m\ )]\!] : T(\mathit{Val}) \\
\quad = \mathit{run}(\mathit{fix}(\lambda(\phi,\gamma).(\textstyle\sum_{i=1}^{n}(\mathcal{P}[\![\mathit{proc}_i]\!]\phi\gamma), \langle\mathcal{A}[\![\mathit{adv}_j]\!]\phi\gamma\rangle_{j=1}^{m}))) \\
\\
\mathit{run}(\phi,\gamma) = \mathcal{E}[\![(\texttt{main})]\!][]\phi\gamma
\end{array}
$$

Here the notation $\langle\ldots\rangle_{j=1}^{m}$ denotes a sequence of length m, and the notation $\sum_{i=1}^{n}$ denotes the concatenation operator on bindings, discussed on page 7.

This completes the semantics of the core language.

5. LESSONS LEARNED

One of the standard claims for formal methods is that that formal methods should 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 7 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

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

Fig. 7. Convoluted fibonacci function

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.

Figure 8 shows a still more convoluted program. This program is a variation on the standard implementation of multiplication as repeated addition, which might be written 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.

```
((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
     (pcalls int loop (int int))
     (args (int x) (int acc))
     (cflow (and
              (pcalls int times (int int))
              (args (int x0) (int y0)))))
   (if (= 0 x)
     acc
     (proceed (- x 1) (+ acc y0))))))
```

Fig. 8. Convoluted multiplication function

5.2 Arguments to `proceed`

As we have seen, `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 procedure, but to the execution of the second advice, as in figure 1. But the variables of an advice body are bound 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 interact 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.

As more possible join points are added, the design possibilities for `proceed` proliferate. For example, in the current (July, 2002) version of ASB, a `proceed` can change the arguments of a method call, but it cannot redirect the call to a different object.

In AspectJ, the `proceed` form takes as arguments the context exposed by the `around`'s pointcut. This is convenient, since in AspectJ, pointcut designators have explicitly declared bound variables, e.g.:

```
(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
  (pcalls int fact (int))
  (args (int y))
  (cflow
    (and
      (pcalls 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. The 2002 version of ASB separates the variables of a pcd into those that may be meaningfully changed and those that may not.

Furthermore, 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. This problem remains unresolved.

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

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

where `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 the 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
  (pcalls 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 [CACM 2001]. AspectJ [Kiczales et al. 2001] is an effort to develop 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.

Andrews [2001] presents a semantics for AOP programs based on a CSP formalism, using CSP synchronization sets as join points. His language 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.

Lämmel [2002] presents static and dynamic operational semantics for a small OO language with a method-call interception facility somewhat different from ours. His system allows dynamic registration of advice, but does not treat `around` advice.

Douence, Motelet, and Sudholt [2001] present an event-based theory of AOP. They present a domain-specific language for defining "crosscuts" (equivalent to our pointcuts). Their language is very powerful, but its semantics is given by a rewriting semantics, which makes the meaning of its programs obscure. We believe that our definition of *match-pcd* represents a significant improvement.

7. FUTURE WORK

We are currently developing a translator from AJD(BASE) to BASE that removes all advice by internalizing the weaving process. We hope to do this in a way that will facilitate a correctness proof.

We plan to extend the ASB suite by adding implementations of the core concepts of other models of AOP and weaving, including static join points, Demeter [Lieberherr 1996], and Hyper/J [Ossher and Tarr 2000]. We hope to develop a theory of AOP that accounts for all of these.

APPENDIX

A. LANGUAGE COMPARISON

Full AJD contains the following features not in the core language captured by the semantics of this paper. None represent difficult extensions for the semantics.

—classes, methods, and objects.

—declared types for bound variables (as illustrated in the examples of section 3).

—static type checking (an `args` pcd includes types for its arguments, as in our examples; at present these must be checked dynamically).

—additional join points at: method calls, method executions, object constructions, field references and field assignments.

—The pcd operators `and` and `or` take an arbitrary number of arguments.

AspectJ provides a sophisticated advice ordering mechanism, where advice is first ordered from most general to most specific, and within classes with equal specificity, orders the advice by qualifier (`before`, `after`, or `around`). AJD is working toward this capability, but the current stable implementation only provides the qualifier-based ordering, where `around` advice is executed around any relevant `before` and `after` advice. In the semantics, advice is ordered by its appearance in the program text.

The examples of section 3 were in written and executed in full AJD except for the following:

—the output was edited to improve formatting

—in the implementation of ASB at the time this work was done, eligible `around` advice was executed in reverse order from its appearance in the program text. The example in figure 1 was edited, reversing the order of advice declarations, to be consistent with the left-to-right semantics of the core language.

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. LNCS 791. Springer-Verlag, 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 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. 17–29.

CACM 2001. *Communications of the ACM*. Vol. 44:10. ACM. special issue on Aspect-Oriented Programming.

Cannon, H. I. 1982. *Flavors: A non-hierarchical approach to object-oriented programming*. Symbolics, Inc.

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.

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, Massachusetts.

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

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 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., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. 1997. Aspect-oriented programming. In *Proceedings 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 *1st International Conference on Aspect-Oriented Software Development*, G. Kiczales, Ed.

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

Moon, D. A. 1986. Object-oriented programming with Flavors. In *Proceedings 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, 2000, Limerick, Ireland*. 734–737.

Smith, B. C. 1984. Reflection and semantics in Lisp. In *Conf. Rec. 11th ACM Symposium on Principles of Programming Languages*. 23–35.

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

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.