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

Mitchell Wand*
College of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA
wand@ccs.neu.edu

Gregor Kiczales and Christopher Dutchyn
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, BC V6T 1Z4, Canada
{gregor,cdutchyn}@cs.ubc.ca

October 9, 2001

Abstract

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.

*Work supported by the National Science Foundation under grant number CCR-9804115.

1 Introduction

A characteristic of aspect-oriented programming, as embodied in AspectJ [KHH+01], 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 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 [Lie96], and Hyper/J [OT00], and to both interpreter-like and compiler-like implementation models.

For more motivation for AOP, see [KLM+97] or the articles [CAC01]. For more on AspectJ, see [KHH+01].

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, the base 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.

So far, this sounds like an instance of the Observer pattern [GHJV95]. 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 outlines the semantics; more details are presented in appendix A. In section 5 we describe some related work, and in section 6 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 of the `proceed` expression executes the actual procedure `fact`, which calls `fact` recursively, which recursively calls 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
```

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.

[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
```

Figure 1: Example of around advice

```
(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
              (pcalls int fact (int))
              (args (int y))
              (cflow
                (and
                  (pcalls int foo (int))
                  (args (int x)))))
      (write x) (write y) (newline))))
```

Figure 2: Binding variables with `cflow`

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` 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. Our intention is that the same base language can then 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 [FWH01].

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

In this abstract, we include only a summary of the semantics; a somewhat more detailed account may be found in appendix A. Definitions of terms may also be found there.

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 visible portion of the program state is defined by:

Join points

jp	$\in$	JP	Join Points
jp	$::=$	$\langle\rangle \mid \langle k, pname, wname, v^*, jp\rangle$	
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 $\langle \texttt{pcall}, f, g, v^*, jp\rangle$ 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.

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

$$
\begin{aligned}
pcd &::= (\texttt{pcalls}\ pname) \mid (\texttt{pwithin}\ pname) \mid (\texttt{args}\ id_1\ \ldots\ id_n) \\
&::= (\texttt{and}\ pcd\ \ pcd) \mid (\texttt{or}\ pcd\ \ pcd) \mid (\texttt{not}\ pcd) \\
&::= (\texttt{cflow}\ pcd) \mid (\texttt{cflowbelow}\ pcd) \mid (\texttt{cflowtop}\ pcd)
\end{aligned}
$$

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*. This function is given by a simple recursion over the structures of pcd's and join points.

Computations are modeled by the monad

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

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.

We can now define the key domains for the semantics:

Domains

χ	$\in$	$T(Val)$	Computations
π	$\in$	$Proc = Val^* \to T(Val)$	Procedures
α	$\in$	$Adv = JP \to Proc \to Proc$	Advice
ϕ	$\in$	$PE = Pname \to Proc$	Procedure Environments
γ	$\in$	$AE = Adv^*$	Advice Environments
ρ	$\in$	$Env = [Id \to Loc] \times WName \times Proceed$	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. The distinguished *Proc* component will be used for the `proceed` operation. We write $\rho(\texttt{\%within})$, $\rho[\texttt{\%within} = \ldots]$, and similarly for `%proceed`, to manipulate these components.

Whenever execution reaches a new join point, the weaver is invoked on the list of all the advice in the program, the new join point, and the code to be executed in that join point (in the semantics, this is always a procedure). The weaver returns a new procedure which 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:

The weaver

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

This brings us to the semantics of advice. 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 is 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). `before` and `after` advice are similar. The details are shown in appendix A.

Semantics of advice declarations

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

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 to produce a computation. Otherwise, it returns the failure computation.

Semantics of pcd’s

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

5 Related Work

Aspect-oriented programming is presented in [KLM+97], which shows how several elements of prior work, including reflection [Smi84], metaobject protocols

[KdR91], subject-oriented programming [HO93], adaptive programming [Lie96], and composition filters [AWB$^+$94] all enable better control over modularization of crosscutting concerns. A variety of models of AOP are presented in [CAC01]. AspectJ [KHH$^+$01] is an effort to develop a Java-based language explicitly driven by the principles of AOP.

Flavors [WM81, Can82], New Flavors [Moo86], CommonLoops [BKK$^+$86] and CLOS [Ste90] all support `before`, `after`, and `around` methods.

Andrews [And01] 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.

Douence, Motelet, and Sudholt [DMS01] 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.

6 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 [Lie96], and Hyper/J [OT00]. We hope to develop a theory of AOP that accounts for all of these.

References

[And01] Jamie H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, 2001. to appear.

[AWB$^+$94] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting Object Interactions Using Composition

Filters. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, LNCS 791, pages 152–184. Springer-Verlag, 1994.

[BKK+86] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: merging Common Lisp and object-oriented programming. In *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 17–29, October 1986.

[CAC01] *Communications of the ACM*, volume 44:10. ACM, October 2001. special issue on Aspect-Oriented Programming.

[Can82] H. I. Cannon. *Flavors: A non-hierarchical approach to object-oriented programming*. Symbolics, Inc., 1982.

[DMS01] Remi Douence, Olivier Motelet, and Mario Sudholt. A formal definition of crosscuts. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, 2001. to appear.

[FWH01] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, second edition, 2001.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1995.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In Andreas Paepcke, editor, *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428. ACM Press, October 1993.

[KdR91] Gregor Kiczales and Jim des Rivieres. *The art of the metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersen, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings European Conference on Object-Oriented Programming*, 2001. to appear.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[Lie96] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[Moo86] D. A. Moon. Object-oriented programming with Flavors. In Norman Meyrowitz, editor, *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–8, New York, NY, November 1986. ACM Press.

[OT00] Harold Ossher and Peri Tarr. 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*, pages 734–737, 2000.

[Smi84] Brian C. Smith. Reflection and semantics in Lisp. In *Conf. Rec. 11th ACM Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[Ste90] Guy L. Steele. *Common Lisp: the Language*. Digital Press, Burlington MA, second edition, 1990.

[WM81] D. Weinreb and D. A. Moon. Flavors: Message passing in the LISP machine. A. I. Memo 602, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, 1981.

A The Semantics in More Detail

Here we give a more detailed account of the semantics. In order to make this presentation self-contained, we duplicate some material that was presented in section 4.

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

$$
\begin{array}{rcll}
v & \in & \mathit{Val} & \text{Expressed Values} \\
l & \in & \mathit{Loc} & \text{Locations} \\
s & \in & \mathit{Sto} & \text{Stores} \\
id & \in & \mathit{Id} & \text{Identifiers (program variables)} \\
\mathit{pname}, \mathit{wname} & \in & \mathit{Pname} & \text{procedure names}
\end{array}
$$

A.1 Join Points

We begin with join points. A join point consists of the following data:

Join points

$$
\begin{array}{rcll}
jp & \in & \mathit{JP} & \text{Join Points} \\
jp & ::= & \langle\rangle \mid \langle k, \mathit{pname}, \mathit{wname}, v^*, jp\rangle & \\
k & ::= & \texttt{pcall} \mid \texttt{pexecution} \mid \texttt{aexecution} & \text{Join Point Kinds}
\end{array}
$$

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 $\langle \texttt{pcall}, f, g, v^*, jp\rangle$ 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.

A.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

$$
\begin{aligned}
pcd ::= & \; (\texttt{pcalls}\ pname) \mid (\texttt{pwithin}\ pname) \mid (\texttt{args}\ id_1\ \ldots\ id_n) \\
::= & \; (\texttt{and}\ pcd\ \ pcd) \mid (\texttt{or}\ pcd\ \ pcd) \mid (\texttt{not}\ pcd) \\
::= & \; (\texttt{cflow}\ pcd) \mid (\texttt{cflowbelow}\ pcd) \mid (\texttt{cflowtop}\ pcd)
\end{aligned}
$$

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*.

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

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

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

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

We can now give the definition of *match-pcd*. The pcd's fall into three groups. 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` $id_1\ \ldots\ id_n$) 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.

match-pcd: **basic operations**

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

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

$$match\text{-}pcd\,(\texttt{args}\ \ id_1\ \ \ldots\ \ id_n)\ \langle k, pname, wname, (v_1, \ldots, v_m), jp\rangle$$
$$= \begin{cases} [id_1 = v_1, \ldots, id_n = v_n] & \text{if } k = \texttt{pcall} \text{ and } n = m \\ \mathit{Fail} & \text{otherwise} \end{cases}$$

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.

Last, we have the temporal operators (`cflow` *pcd*), (`cflowbelow` *pcd*), and (`cflowtop` *pcd*). The pcd (`cflow` *pcd*) finds the latest 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.

match-pcd: **temporal operators**

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

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

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

A.3 The Execution Monad

To package the execution, we introduce a monad:

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

This is a monad with three effects: a dynamically-scoped quantity of type *JP*, a store of type *Sto*, and 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

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

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

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

The lift operation induces an order on $T(A)$ for any A. In general, we will use Greek letters for metavariables ranging over partial orders and Roman letters for metavariables ranging over sets. We will use the following domains:

Domains

χ	$\in$	$T(Val)$	Computations
π	$\in$	$Proc = Val^* \rightarrow T(Val)$	Procedures
α	$\in$	$Adv = JP \rightarrow Proc \rightarrow Proc$	Advice
ϕ	$\in$	$PE = Pname \rightarrow Proc$	Procedure Environments
γ	$\in$	$AE = Adv^*$	Advice Environments
ρ	$\in$	$Env = [Id \rightarrow Loc] \times WName \times Proceed$	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. The distinguished *Proc* component will be used for the `proceed` operation. We write $\rho(\texttt{\%within})$, $\rho[\texttt{\%within} = \ldots]$, and similarly for `%proceed`, to manipulate these components.

A.4 Expressions

We can now give the semantics of expressions. We give here only a fragment:

Semantics of expressions

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

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

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

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.

A.5 *enter-join-point*, 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^*.

enter-join-point

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

The weaver is the heart of the system. It takes a list of advice, a join point, and a procedure, and it returns a procedure. It 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, passing as its `%proceed` the procedure that results from weaving the rest of the advice. Thus

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

The weaver

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

This brings us to the semantics of advice. 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, 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.

Semantics of advice

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

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

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

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

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

A.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 are bound to the new locations.

Semantics of procedure declarations

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

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

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

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

Here the notation $\langle\ldots\rangle_{j=1}^{m}$ denotes a sequence of length m, and the notation $\sum_{i=1}^{n}$ refers to the concatenation operator on bindings, discussed on page 15.

This completes the semantics of the core language.

B 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 current implementation, eligible `around` advice is 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.