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

joint work with Gregor Kiczales and Christopher Dutchyn
(Univ. of British Columbia)

September, 2001

What is Aspect-Oriented Programming?

- Some concerns are “crosscutting”
- They appear over and over again in the code
- Examples: logging, security, synchronization, serialization, persistence
- Gregor calls these *aspects*

Modularizing Aspects

- Put policy decisions (code) together in one place
- Then make sure the functionality is installed in all the places it belongs
- Nothing new so far; could do this with modules/objects/etc and procedure/method calls
- But installation instructions are still distributed
- Worse yet, may want to trigger new functionality *dynamically*

Modularizing Aspects, cont'd

- *New Twist:* Use a declarative language of *pointcut descriptors* to specify where the functionality is to go.

- Now functionality and installation directions are modularized together

- *Join Point Model:* specifies where/how the functionality can be installed (*woven*) into the program

- A Join Point Model can be *static* or *dynamic*

Static Join Point Models

- A join point is a place in the program text
- Possible actions at a join point: add/substitute code, add new variables, etc.
- Design decision: which points in the text are potential join points?
- Design decision: how to specify a set of places?
- Examples: HyperJ [Tarr & Ossher] , Aspects/Collaborations [Lieberherr & Ovlinger]

Dynamic Join Point Models

- A join point is a point in the execution of the program
- Possible actions at a join point: execute code before/after/around (instead of) original code
- Examples: AspectJ, Demeter

Dynamic Join Point Models, cont'd

- Design decision: which points in the execution are potential join points?

- Design decision: how to specify a set of join points?

- Design decision: how to communicate data from program to aspect code?

Comparing Static and Dynamic Join Points

- Static and Dynamic join point models are *incomparable*
- Can implement some, but not all dynamic join points statically
- Can do some things statically that don't make sense dynamically (e.g., add variables)

More Applications

- Can use this technology for any functionality that crosses natural module borders, or even within a module

- Demeter: computations are tree searches, join points are traversals of nodes, specified by type (Lieberherr)

- AIRES: computations are remote method calls, join points are passages through protocol layers; actions are distributed through the layers and across hosts. (joint work with Lieberherr and BBN)

Aspect Sandbox (ASB)

- Goal is to have a minilanguage for playing with these ideas
- Language: recursion equations + assignment + simple classes & objects
- Want to experiment with different join point models and implementations of weaving
- We'll concentrate on dynamic join points, no types, no objects. This language is called AJD/PROC

Rest of the Talk

- Examples using dynamic join point model

- Some semantics:
  - Join points and point-cut designators
  - Denotational semantics for the language

    Monadic style

Advice

- Aspects are written using *advice*. Typical advice:

$$\texttt{(around } \mathit{pcd}\ \ e\texttt{)}$$

- *pcd* is a pointcut designator (grammar to follow)

- specifies at which join points e is to be executed

- also binds variables in e

Advice, cont'd

(`around` *pcd* *e*)

- *e* is executed *instead* of the code at the join point
- can execute the underlying code using `proceed` expression
- Also have `before` and `after` advice, with implicit `proceed`

Example

```
((procedure void main () (write (fact 3)))
 (procedure int fact ((int x))
   (if (< x 1) 1
     (* x (fact (- x 1)))))
 (around
   (and
     (pcalls int fact (int))
     (args (int n)))
   (write 'before) (write n)
   (let (((int ans) (proceed n)))
     (write 'after) (write ans)
     ans))))
```

prints:

```
before 3
before 2
before 1
before 0
after 1
after 1
after 2
after 6
6
```

```
((procedure void main () (write (+ (fact 5) (foo 3))))
 (procedure int foo ((int x)) (fact x))
 (procedure int fact ((int x))
   (if (< x 1) 1
     (* x (fact (- x 1)))))
 (around
   (and
     (pcalls int fact (int))
     (cflow (pcalls int foo (int))) ;; <--
     (args (int n)))
   (write 'before) (write n)
   (let (((int ans) (proceed n)))
     (write 'after) (write ans)
     ans)))
```

prints:

```
before 3
before 2
before 1
before 0
after 1
after 1
after 2
after 6
126
```

Fun with proceed

```
((procedure void main () (fib 19))
 (procedure int fib ((int n)) (fib n))

 (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))))))
```

Semantics

- Syntax and semantics of join points and pcd's
  - Need to describe the data exported at a join point
  - Need to describe which join points match which pcd's

- Semantics of the minilanguage

Join points

$$
\begin{array}{lll}
jp & ::= & \langle\,\rangle \qquad\qquad\qquad\qquad \text{Top of Program} \\
 & ::= & \langle \mathtt{pcall}, pname, pname', v^*, jp \rangle \\
 & & \quad \text{Call of procedure } pname \text{ from within } pname\text{'} \\
 & & \quad \text{with arguments } v^* \text{ in context } jp \\
 & ::= & \langle \mathtt{pexecution},\ jp \rangle \qquad \text{Entering a procedure body} \\
 & ::= & \langle \mathtt{aexecution},\ jp \rangle \qquad \text{Entering an advice body}
\end{array}
$$

Pointcut designators

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

A pcd matched against a join point either returns a set of bindings or fails.

Algebra of pcd results

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

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

($b + b'$ denotes concatenation of bindings)

match-pcd: **basic operations**

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

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

Other join points fail

match-pcd: binding arguments

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

$\mathit{match\text{-}pcd}$: temporal operators

$$\mathit{match\text{-}pcd}\,(\texttt{cflow}\ \mathit{pcd})\,\langle\,\rangle = \mathit{Fail}$$

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

$$\mathit{match\text{-}pcd}\,(\texttt{cflow}\ \mathit{pcd})\,\langle k,\ \mathit{jp}\rangle \qquad (k = \texttt{pexecution},\ \texttt{aexecution})$$
$$= \mathit{match\text{-}pcd}\ \mathit{pcd}\,\langle k,\ \mathit{jp}\rangle \vee \mathit{match\text{-}pcd}\,(\texttt{cflow}\ \mathit{pcd})\,\mathit{jp}$$

Integrating this into the language

We'll use a monad

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

The monad **let** and **return** ensure that *JP* has dynamic scope, and that *Sto* is global

Monad operations

$$\textbf{return}\ v = \lambda\, jp\ s\,.\, lift(v, s)$$

$$\textbf{let}\ v \Leftarrow E_1\ \textbf{in}\ E_2$$

$$\begin{aligned} = \lambda\, jp\ s\,.\,\textbf{case}\ (E_1\ jp\ s)\ \textbf{of}\ & \\ & \bot \Rightarrow \bot \\ & lift(v) \Rightarrow ((\lambda\, v\,.\,E_2)\ v\ jp\ s) \end{aligned}$$

Domains

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

Semantics of expressions

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

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

`proceed`

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

enter-join-point

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

enter-join-point γ f π v^* is 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^*.

setjp

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

The weaver

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

So we have

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

The Meaning of Advice

$$Adv = JP \rightarrow Proc \rightarrow Proc$$

An advice is a conditional procedure transformer: if the join point matches, it wraps itself around a procedure (binding `%proceed` along the way), otherwise it returns the procedure unchanged.

The weaver asks each advice in turn to wrap itself around the current procedure in the current join point.

Semantics of advice declarations

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

Semantics of procedure declarations

$$
\begin{array}{l}
P[\![(\texttt{procedure}\ \mathit{pname}\ (x_1\ \ldots\ x_n)\ e)]\!] : PE \rightarrow AE \rightarrow PE \\
\quad = \lambda\phi\gamma.[\mathit{pname} = \\
\qquad\qquad \lambda v^*.(\mathit{enter\text{-}join\text{-}point}\ \gamma \\
\qquad\qquad\qquad (\mathit{new\text{-}pexecution\text{-}jp}\ \ \mathit{pname}) \\
\qquad\qquad\qquad (\lambda w\,.\ \textbf{let}\ l_1 \Leftarrow \textbf{alloc}(w{\downarrow}1)\ ;\ldots;\ l_n \Leftarrow \textbf{alloc}(w{\downarrow}n) \\
\qquad\qquad\qquad\qquad \textbf{in}\,(E[\![e]\!][\texttt{\%within} = \mathit{pname},\ x_1 = l_1,\ \ldots,\ x_n = l_n]\ \phi\ \gamma)) \\
\qquad\qquad\qquad v^*)]
\end{array}
$$

Semantics of programs

$$PGM[\![(\mathit{proc}_1\ \ldots\ \mathit{proc}_n\ \mathit{adv}_1\ldots\ \mathit{adv}_m)]\!] : T(\mathit{Val})$$

$$= \mathit{run}(\mathit{fix}(\lambda(\phi,\gamma).(\textstyle\sum_{i=1}^{n}(P[\![\mathit{proc}_i]\!]\phi\gamma),\ \langle A[\![\mathit{adv}_j]\!]\phi\gamma\rangle_{j=1}^{m})))$$

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

A program is a set of mutually recursive procedure and advice declarations. We run a program by calling the procedure `main` on no arguments.

Summary

- We've given a brief introduction to AOP, static vs. dynamic join point models, and sketched some applications

- We've given some examples of dynamic join points and advice in AJD/PROC

- We've shown the semantics of join points and pcd's in AJD/PROC (simple!)

- We've shown the semantics of programs in AJD/PROC (maybe not so simple!)

What's next?

- Applications (AIRES, Demeter)
- Scaling up (types, classes, objects)
- Program Analysis (CFA, SBA)
- Other join point models (static join points)
- Compiling AJD/PROC via phase separation (!)