Is Join Point a Point?
a pointcut and advice mechanism for making aspects more reusable

Hidehiko Masuhara
University of Tokyo

joint work with Yusuke Endoh and Aki Yonezawa
A secret of AspectJ

- AspectJ is based on *join points*
- A secret of AspectJ:
  
  join points are *not* points, but *regions*

  - makes aspects hard to maintain

- We propose an AOPL
  
  in which join points are *points*
Aspect-oriented programming enables modularization of crosscutting concerns [Kiczales et al. 1997] e.g., logging, security, error handling offers a mechanism: join points, pointcut & advice
Join point, pointcut & advice in AspectJ

Base: reads user’s inputs in several modules
Logging: record all inputs from the console
  • Join points=actions like method calls, execs...
  • Pointcut: selects jps
  • Advice: runs on selected jps

```java
after() returning(String x) :
  call(String * .readLine()) {
    Logger.log(x);
  }
```
Aspect maintainability: most changes are adapted by pointcuts

Changes in aspect spec./base prog. ↓

after() returning(String x) :
  \(\text{call(String *.*readLn())} \lor \text{call(String *.*getenv())} \land \neg \text{within(LogBrowser)}\) {
  Logger.log(x);
}

• log getenv as well
• exclude calls from LogBrowser
• rename readLine to readLn
• log onSubmit as well

\uparrow \text{modifications to pointcuts to cope with changes}
Aspect maintainability: some changes can not be adapted by pointcuts

Logging inputs from console & GUI widgets needs two advice decls. i.e., can not be adapted by pointcuts

```java
after() returning(String x) :
    call(String *.readLine()) {
        Logger.log(x);
    }
before(String x):
    exec(* *.onSubmit(String)) && args(x) {
        Logger.log(x);
    }
```
Another example: returning null vs. throwing exceptions

```java
r = find(...);
if (r==null)
    handle not found case
process the result
```

```java
try {
    r = find(...);
} catch (NotFound e) {
    handle not found case
}
process the result
```

```java
after() returning (Result r):
call(find) && if(r==null) {
    Logger.log();
}
```

```java
after() throwing (NotFound):
call(find) {
    Logger.log();
}
```
Problem summary & analysis

- Generalization: can not advise “beginnings of X and ends of Y” by one decl.
  - active / passive parameter passing
  - returning error values / throwing exceptions
  - direct style / continuation passing style (in FPL)

- Reasons:
  - join points are regions w/ entry and exit
  - pointcuts select only join points
  - advice decls. specify entry or exit
Proposal: AOP mechanism based on \textit{point-in-time} join points

- Overview
- Aspect maintainability with point-in-time join points
- Design issues of pointcuts and advice
- Formalization
AOP mechanism based on *point-in-time* join points

- A join point is a point in time
- New join points that represent ends of actions
- New pointcuts that select new join points

```
advice(String x) : 
    return(String * .readLine())
    && args(x) {
        Logger.log(x); proceed x;
    }
```

```
Main
    call jp
    readLine
        return jp
        "john"
    Console
```
Aspect maintainability with point-in-time join points: *logging readLine & onSubmit*

- One advice decl. can log both
  - return values from readLine
  - parameters to onSubmit

```java
advice(String x) :
  (return(String *.readLine())
   || call(void *.onSubmit(String x)))
  && args(x) {
    Logger.log(x); proceed x;
  }
```
Aspect maintainability with point-in-time join points:

returning null vs. throwing exceptions

r = find(...);
if (r==null)
   handle not found case
   do with the result

try {r = find(...);}
catch (NotFoundException e) {
   handle not found case
}
do with the result

advice(): (return(find) &&
   args(r) && if(r==null)) ||
throw(find,NotFoundException) {
   Logger.log();proceed;
}

one advice decl.
for two
Design issues of pointcuts & advice with point-in-time join points

- Designed to support most features in AspectJ
  - run code before or after a jp
  - replace parameters to a jp
  - replace a return value from a jp
  - skip execution of a jp
- ...but difficult to support some:
  - repeat execution of a jp
  - run code before and after a jp

```java
void around(String x) {
    proceed(x.lower());
}
```

- introduced a special form to pass new params to jp
- introduced another form to skip to the caller

```java
return(readLine()) && args(x) {
    proceed x.lower() ;
}
```

convert parameters to onSubmit into lowercase

```java
convert parameters to onSubmit into lowercase
```

```java
introduced a special form to pass new params to jp
```

```java
introduced another form to skip to the caller
```

```java
proceed to a return jp
```

```java
proceed to a return jp
```
Design issues of pointcuts & advice with point-in-time join points

- Designed to support most features in AspectJ
  - run code before or after a join point
  - replace parameters to a join point
  - replace a return value from a join point
  - skip execution of a join point

...but difficult to support some:
- repeat execution of a join point
- run code before **and** after a join point

proceed won’t come back in point-in-time
Formalization of pointcut & advice based on point-in-time join points

Writing a denotational semantics
- of an untyped FPL + pointcut&advice
- by using a continuation passing style (CPS)
  - a return = application to a continuation
- simpler in terms of advice exec.
  - no longer has specifiers like “before”
- suitable to explore advanced features
  - e.g., advising exceptions
Semantics of advice execution: a sample session

An expression:
\[
\text{let } f(x) = x + x \text{ in } f(1)
\]

with advice:
\[
\begin{align*}
\text{advice}(x) : \text{call}(f) & \{ \text{proceed } x + 1; \} \\
\text{advice}(x) : \text{return}(f) & \{ \text{proceed } x / 2; \}
\end{align*}
\]

Execution trace:
1. creates a jp “call f with 1”
2. matches pointcut “call(f)”
3. evaluates “proceed x+1”
4. calls f with 2
5. creates a jp “return from f with 4”
6. matches pointcut “return(f)”
7. evaluates “proceed x/2”
8. yields 2
Semantics of advice execution: function call w/o advice

- semantic function
  \[ E : \text{Exp} \rightarrow \text{Env} \rightarrow \text{Ctn} \rightarrow \text{Ans} \]
  \[ \text{Ctn} = \text{Val} \rightarrow \text{Ans} \]
  \[ E[(E_0 \ E_1)] \rho \ \kappa = E[E_0] \rho \]
  \[ (\lambda f. \ E_1 \rho (\lambda v. f (\lambda v'. \kappa v') v)) \]

A function is denoted by a term of type \( \text{Ctn} \rightarrow \text{Val} \rightarrow \text{Ans} \)
Semantics of advice execution: function call with advice

- semantic function
  \[ E : \text{Exp} \rightarrow \text{Env} \rightarrow \text{Ctn} \rightarrow \text{Ans} \]
  \[ \text{Ctn} = \text{Val} \rightarrow \text{Ans} \]
  \[ E[(E_0, E_1)] \rho \kappa = E[E_0] \rho (\lambda f. E[E_1] \rho (\lambda v. \ldots)) \]

\( \lambda v'. W A \theta' \kappa \nu' \)

- can treat call & return jps uniformly
- weaver advice decls.
- jp “call f”
- jp “return f”
Semantics of advice execution: weaver

- $W : Adv \rightarrow Jp \rightarrow Ctn \rightarrow Ctn$

$W [\text{advice}(x) : p \{E\}] \theta \kappa v =$
  if $p$ matches $\theta$
    then $E [E] [v/x] \kappa$
  else $\kappa v$
Semantics of advanced features (ongoing)

- Uniform representation of *exception throwing* mechanisms
  - represents exception handlers as continuations
  - creates “throw” join point at throwing exceptions
- Support for *history sensitive pointcuts*
  - similar approach to tracecuts [Walker00]
  - would subsume cflow
- Interaction with *tail call elimination*
  - crucial in FPL
  - folding eta-expanded continuations
Related work: extension to pointcuts and advice

• Poincuts that capture return values:
  \texttt{dflow} [APLAS’03], \texttt{Arachne} [Douence’05]
  – based on region-in-time join points

• Fine grained jps:
  \texttt{LoopsAJ} [Harbulot’05], \texttt{Eos-T} [Rajan’05],
  \texttt{bugdel} [Usui’05]
  – based on region-in-time join points
Related work:

- **Aspect SandBox** [Wand’02]
  - region-in-time, denotational & direct style
  - semantic function for each of before/after/around

\[
\begin{align*}
\mathcal{A}[\text{around } \text{pcd } e] \phi \gamma &: JP \to Proc \to Proc \\
&= \lambda jp \, \pi \, v^* \cdot \mathcal{P} \mathcal{C} \mathcal{D}[\text{pcd}] \![j]p \\
&\quad (\lambda p. \text{enter-join-point } \gamma \, \text{new-execution-jp} \\
&\quad (\lambda v^* \cdot \mathcal{E}[\gamma](\rho[\%\text{within} = \text{None}, \%\text{proceed} = \pi])\phi)) \\
&\quad \langle) \\
&\quad (\pi \, v^*)
\end{align*}
\]

\[
\begin{align*}
\mathcal{A}[\text{before } \text{pcd } e] \phi \gamma &: JP \to Proc \to Proc \\
&= \lambda jp \, \pi \, v^* \cdot \mathcal{P} \mathcal{C} \mathcal{D}[\text{pcd}] \![j]p \\
&\quad (\lambda p. \text{enter-join-point } \gamma \, \text{new-execution-jp} \\
&\quad (\lambda v^* . \text{let } v_1 \leftarrow \mathcal{E}[\gamma](\rho[\%\text{within} = \text{None}, \%\text{proceed} = \text{None}]\phi) \\
&\quad \quad v_2 \leftarrow (\pi \, v^*) \\
&\quad \text{in } v_2) \\
&\quad \langle) \\
&\quad (\pi \, v^*)
\end{align*}
\]

\[
\begin{align*}
\mathcal{A}[\text{after } \text{pcd } e] \phi \gamma &: JP \to Proc \to Proc \\
&= \lambda jp \, \pi \, v^* \cdot \mathcal{P} \mathcal{C} \mathcal{D}[\text{pcd}] \![j]p \\
&\quad (\lambda p. \text{enter-join-point } \gamma \, \text{new-execution-jp} \\
&\quad (\lambda v^* . \text{let } v_1 \leftarrow (\pi \, v^*) \\
&\quad \quad v_2 \leftarrow \mathcal{E}[\gamma](\rho[\%\text{within} = \text{None}, \%\text{proceed} = \text{None}]\phi) \\
&\quad \text{in } v_1) \\
&\quad \langle) \\
&\quad (\pi \, v^*)
\end{align*}
\]
Final remarks: a pointcut & advice mechanism based on point-in-time join points

• **Design**
  – can uniformly treat beginnings and ends of actions
  – some missing features (eg repeating jps)

• **Semantics**
  – in a continuation passing style
  – uniformly treat calls and returns
  – advanced features (eg exception, cflow, TCE)

• **Implementation**
  – development of a compilation model

• **Evaluation**
  – assesement of aspect maintainability