Pointcuts and Advice for Higher-Order Languages

David B. Tucker and Shriram Krishnamurthi

2003 March 20
An example

parse  makeBackup  prettyPrint

readContents  writeContents

openFile  closeFile
An example with aspects

Possible aspects:

- trace calls to closeFile originating from makeBackup
- check for legal arguments to writeContents
- ensure the callee has permission to execute openFile

Will show how to define such aspects in a higher-order language
Why AOP in a higher-order language?

- Many languages have higher-order first-class functions
  - Scheme, ML, Haskell
Why AOP in a higher-order language?

• Many languages have higher-order first-class functions
  ★ Scheme, ML, Haskell
  ★ Perl, Python, Ruby
Why AOP in a higher-order language?

- Many languages have higher-order first-class functions
  - Scheme, ML, Haskell
  - Perl, Python, Ruby

- What is interaction between FP and AOP?
  - simplify specification of aspects?
  - define more general aspects?
Challenges

• How to specify aspects
  ★ a function may have zero, one, or multiple names
  ★ first-order or first-class aspects?
Challenges

• How to specify aspects
  ★ a function may have zero, one, or multiple names
  ★ first-order or first-class aspects?

• Scoping issues
  ★ can define aspects outside top level
  ★ when is an aspect in force?
Challenges

- How to specify aspects
  - a function may have zero, one, or multiple names
  - first-order or first-class aspects?

- Scoping issues
  - can define aspects outside top level
  - when is an aspect in force?

Will present extension of a higher-order language that supports pointcuts and advice
How to specify?

Decided to make pointcuts and advice first-class

- Consistent with design of functional languages
- Define pcd as predicate over list of join points
- Define advice as join point (procedure) transformer
How to specify pcd’s?

Calls to $closeFile$

In AspectJ:

```java
call( void closeFile() )
```
How to specify pcd’s?

Calls to closeFile

In AspectJ:

\[
\text{call(}\text{void closeFile()}\text{)}
\]

In our language:

\[
(\lambda \ (jpl)
\quad (\text{eq? close-file (first jpl})))))
\]
How to specify pcd’s?

Calls to \textit{closeFile} originating from \textit{makeBackup}

In AspectJ:

\begin{verbatim}
call(void closeFile()) && cflow(withincode(void makeBackup()))
\end{verbatim}
How to specify pcd’s?

Calls to closeFile originating from makeBackup

In AspectJ:

```
call(void closeFile())
&& cflow(withincode(void makeBackup()))
```

In our language:

```
(\ (jpl)
  (and (eq? close-file (first jpl))
       (member make-backup (rest jpl))))
```
How to specify pcd’s?

\((\text{call } f) \equiv (\lambda (jpl) (eq? f (\text{first } jpl)))\)
How to specify pcd’s?

\[
\text{(call } f) \equiv (\lambda \ (jpl) \ (eq? \ f \ (first \ jpl)))
\]

\[
\text{(within } f) \equiv (\lambda \ (jpl) \ (\text{and} \ (not \ (empty? \ (rest \ jpl))))
\]

\[
(eq? \ f \ (second \ jpl)))
\]
How to specify pcd’s?

\[(\text{call } f) \equiv (\lambda (jpl) (eq? f (\text{first } jpl)))\]
\[(\text{within } f) \equiv (\lambda (jpl) (\text{and} (not (\text{empty?} (\text{rest } jpl)))) (eq? f (\text{second } jpl))))\]
\[(\&\& \ pcd1 \ pcd2) \equiv (\lambda (jpl) (\text{and} (pcd1 jpl) (pcd2 jpl))))\]
How to specify pcd's?

(call f) ≡ (\( \lambda \) (jpl) (eq? f (first jpl)))

(within f) ≡ (\( \lambda \) (jpl) (and (not (empty? (rest jpl)))

(eq? f (second jpl))))

(\&\& pcd1 pcd2) ≡ (\( \lambda \) (jpl) (and (pcd1 jpl)

(pcd2 jpl))))

(cflow pcd) ≡ (\( \lambda \) (jpl)

(cond

[(empty? jpl) false]

[else (or (pcd jpl)

(((cflow pcd) (rest jpl)))))]]))
How to specify pcd’s?

Rewrite examples as:

Calls to closeFile

(call close-file)
How to specify pcd’s?

Rewrite examples as:

Calls to closeFile
(call close-file)

Calls to closeFile originating from makeBackup
(&& (call close-file) (cflow (within make-backup)))
How to specify pcd’s?

Rewrite examples as:

Calls to closeFile
  (call close-file)

Calls to closeFile originating from makeBackup
  (&& (call close-file) (cflow (within make-backup)))

Showed how to define pcd’s

Next: how to define advice
How to specify advice?

Procedure transformers:

\[
\text{(define trace-advice}
  \ (\lambda \ (\text{proc})
  \ (\lambda \ (\text{arg})
  \ \quad \ (\text{printf } \text{"calling open-file"})
  \ \quad \ \text{(proceed proc arg)})))
\]
How to specify advice?

Procedure transformers:

\[
\text{(define trace-advice}
\begin{align*}
&\ (λ\ (proc) \\
&\quad (λ\ (arg) \\
&\quad\quad (\text{printf } "\text{calling open-file"}) \\
&\quad\quad (\text{proceed}\ proc\ arg))))
\end{align*}
\]

All advice is \textbf{around} advice
How to specify advice?

Procedure transformers:

(define trace-advice
  (λ (proc)
    (λ (arg)
      (printf "calling open-file")
      (proceed proc arg))))

All advice is around advice

So far, no more or less than AspectJ
The around expression

To install a pcd and advice, introduce new type of expression:

\[(\text{around } Pcd \text{ Advice Body})\]
The around expression

To install a pcd and advice, introduce new type of expression:

(around Pcd Advice Body)

For example:

(let ([input (parse "file1")])
  (around (call open-file) trace-advice
    (pretty-print input "file2"))))
Review of scope in Java

interface StringMaker {
    String process(String s);
}
Review of scope in Java

```java
interface StringMaker {
    String process(String s);
}

final String familyName = "tucker";

StringMaker makeFamilyMember = new StringMaker() {
    String process(String givenName) {
        return givenName + " " + familyName;
    }
};
```
interface StringMaker {
    String process(String s);
}

final String familyName = "tucker";
StringMaker makeFamilyMember = new StringMaker() {
    String process(String givenName) {
        return givenName + " " + familyName;
    }
};

makeFamilyMember.process("dave")
Review of scope in Java

```java
interface StringMaker {
    String process(String s);
}

final String familyName = "tucker";

StringMaker makeFamilyMember = new StringMaker() {
    String process(String givenName) {
        return givenName + " " + familyName;
    }
};

makeFamilyMember.process("dave")
⇒ "dave tucker"
```
Review of scope in Java

What happens here...?

```java
final String familyName = "krishnamurthi";
makeFamilyMember.process("dave");
```
Review of scope in Java

What happens here...?

```
final String familyName = "krishnamurthi";
makeFamilyMember.process("dave");
```

*Static* scoping (Java) ⇒ "dave tucker"

- *familyName*’s value from site of function *definition*
Review of scope in Java

What happens here...?

```java
final String familyName = "krishnamurthi";
makeFamilyMember.process("dave");
```

**Static** scoping (Java) ⇒ "dave tucker"

- `familyName`'s value from site of function *definition*

**Dynamic** scoping ⇒ "dave krishnamurthi"

- `familyName`'s value from site of function *application*
What is scope for aspects?

In AspectJ, aspects defined in top-level scope, and apply to everything in that scope.
What is scope for aspects?

In AspectJ, aspects defined in top-level scope, and apply to everything in that scope

In a higher-order language, can define aspects more precise scopes
What is scope for aspects?

In AspectJ, aspects defined in top-level scope, and apply to everything in that scope.

In a higher-order language, can define aspects more precise scopes.

**around** aspects are *statically* scoped.

- apply to join points in *text* of body.
Example #1

(around (call open-file) trace-advice (open-file "boston"))
Example #1

(around (call open-file) trace-advice (open-file "boston"))

This prints a trace message
Example #2

```((around (call open-file) trace-advice
  (λ (f) (open-file f)))
"boston")
```
Example #2

\[
\begin{align*}
((\textbf{around} \ (\textit{call open-file}) \ \textit{trace-advice})
(\lambda (f) \ (\textit{open-file} f)))
\quad "boston"
\end{align*}
\]

Also prints a trace message
Example #3

(let ([apply-to-boston (λ (f) (f "boston"))])
(around (call open-file) trace-advice
  (apply-to-boston open-file)))
Example #3

(let ([apply-to-boston (\(f\) (f "boston"))]))
(around (call open-file) trace-advice
(apply-to-boston open-file)))

This *does not* print a trace message
Example #3 revisited

Can we define aspects that do apply?

(let ([apply-to-boston (λ f (f "boston"))])))
(around (call open-file) trace-advice
 (apply-to-boston open-file)))
Example #3 revisited

Can we define aspects that do apply?

(let ([apply-to-boston (λ (f) (f "boston")))]
  (fluid-around (call open-file) trace-advice
    (apply-to-boston open-file))))
Example #3 revisited

Can we define aspects that do apply?

\[
\text{(let ([apply-to-boston (λ (f) (f "boston"))])}
\text{(fluid-around (call open-file) trace-advice}
\text{(apply-to-boston open-file))})
\]

This does print a trace message

**fluid-around** aspects are *dynamically* scoped

- apply to join points during *evaluation* of body
Example #2 revisited

(((around (call open-file) trace-advice
  (λ (f) (open-file f)))
"boston")
Example #2 revisited

\[
\left( \textbf{fluid-around} \ (\textit{call open-file}) \ \textit{trace-advice} \ \\
(\lambda \ (f) \ (\textit{open-file} \ f)) \ \\
"boston" \right)
\]
Example #2 revisited

```scheme
((fluid-around (call open-file) trace-advice
   (λ (f) (open-file f)))
"boston")
```

Does not print a message
Using dynamic aspects

Trace calls to *close-file* that originate from *make-backup*
Using dynamic aspects

Trace calls to close-file that originate from make-backup

(define (backup-system)
  (for-each make-backup
    (list "boston" "providence" "woonsocket")))
Using dynamic aspects

Trace calls to close-file that originate from make-backup

\[
\begin{align*}
  \text{define} & \quad (backup\text{-}system) \\
  & \quad (\text{for\text{-}each} \ \text{make\text{-}backup} \\
  & \quad \quad (\text{list} \ "\text{boston}" \ "\text{providence}" \ "\text{woonsocket}")) \\
  & \quad (\text{fluid\text{-}around} \ (\&\& \ (\text{call close\text{-}file}) \\
  & \quad \quad (\text{cflow (within make\text{-}backup})) \\
  & \quad \quad \text{trace\text{-}advice} \\
  & \quad \quad (backup\text{-}system))
\end{align*}
\]
Using static aspects

Ensure the callee has permission to execute `openFile`

Use stack inspection to check privileges:

- a trusted user must ask for privilege
- if privilege is on stack, with no intervening untrusted code, then go ahead

Concisely: “only trusted frames UNTIL privilege granted”
Using static aspects

Easy:

\[(\text{define } \text{protected-open-file} \\ \text{(around } (\&\&\& \text{(call open-file)})
\quad (! \text{(until trusted? privileged?)})
\text{report-privilege-error})\)
\]

\[(\lambda (f)
\quad \text{(open-file } f)))\]

Can export this function
Higher-order pointcuts

Since pointcuts are first-class, we could define \textit{until}:

\[
\text{(define \ (until pcd1 pcd2)} \\
\quad (\lambda \ (jpl) \\
\quad \quad (\text{cond} \\
\quad \quad \quad [(\text{empty?} \ jpl) \ \text{false}] \\
\quad \quad \quad [\text{else} \ (\text{or} \ (pcd2 \ jpl) \\
\quad \quad \quad \quad (\text{and} \ (pcd1 \ jpl) \\
\quad \quad \quad \quad \quad (\text{until} \ pcd1 \ pcd2) \ (\text{rest} \ jpl))))))))
\]

Can you write this using \texttt{cflow}?
Implementation background

- Hygienic macros (*syntax-case*)
- PLT Scheme module system
- Continuation marks:
  - *(w-c-m Tag Value Body)* adds a mark
  - *(c-c-m Tag)* retrieves marks
Continuation marks example

For example:

(\textbf{define} (\textit{fact} \textit{n})
    (\textbf{w-c-m} \textit{fact-arg} \textit{n}
     (\textbf{if} (\textit{zero?} \textit{n})
         (begin (\textbf{display} (\textbf{c-c-m} \textit{fact-arg})) 1)
         (* \textit{n} (\textit{fact} (\textbf{sub1} \textit{n})))))))
Continuation marks example

(fact 2)
Continuation marks example

\((fact\ 2)\)

\(\Rightarrow (w-c-m\ \text{'}fact-arg\ 2\)
\(\quad \times\ 2\)
\(\quad (w-c-m\ \text{'}fact-arg\ 1\)
\(\quad \times\ 1\)
\(\quad (w-c-m\ \text{'}fact-arg\ 0\)
\(\quad \text{begin}\ (display\ (c-c-m\ \text{'}fact-arg))\ 1)))))))\)
Continuation marks example

\[(\text{fact } 2)\]

\[\Rightarrow (\text{w-c-m } \text{'}\text{fact-arg } 2
\hspace{1em}(* \hspace{1em} 2
\hspace{1em} (\text{w-c-m } \text{'}\text{fact-arg } 1
\hspace{1em} (* \hspace{1em} 1
\hspace{1em} (\text{w-c-m } \text{'}\text{fact-arg } 0
\hspace{1em} (\text{begin } (\text{display } (\text{c-c-m } \text{'}\text{fact-arg})) 1))))))))\]

displays \((0 \hspace{1em} 1 \hspace{1em} 2)\)
Implementation of dynamic aspects

- Join points
  - record with (\texttt{w-c-m } 'joinpoint \texttt{fun-val} \ldots )
  - retrieve current list with (\texttt{c-c-m } 'joinpoint)
Implementation of dynamic aspects

- Join points
  - record with \((w\text{-}c\text{-}m ~ 'joinpoint ~ fun\text{-}val ~ . . . )\)
  - retrieve current list with \((c\text{-}c\text{-}m ~ 'joinpoint)\)

- Dynamic aspects
  - \textbf{fluid\text{-}around} does \((w\text{-}c\text{-}m ~ 'dynamic ~ aspect ~ . . . )\)
  - application retrieves aspects with \((c\text{-}c\text{-}m ~ 'dynamic)\)
Implementation of dynamic aspects

- Join points
  - record with (w-c-m 'joinpoint fun-val . . . )
  - retrieve current list with (c-c-m 'joinpoint)

- Dynamic aspects
  - fluid-around does (w-c-m 'dynamic aspect . . . )
  - application retrieves aspects with (c-c-m 'dynamic)

- Function application has list of joinpoints and dynamic advice, can invoke aspects (similar to semantics)
Implementation of static aspects

- Transform all lambdas to remember active aspects
- When applied, functions automatically reinstate static aspects
- Make sure to use correct aspects during function application
Limitations

AspectJ can match data on any join point in context:

```
pointcut factArg(int n) :
    call(int fact(int)) && args(n);
```
Limitations

AspectJ can match data on any join point in context:

```java
pointcut factArg(int n) :
    call(int fact(int)) && args(n);

before(int x, int y) :
    factArg(x) && cflowbelow(factArg(y))
{
    System.out.println(x + " " + y);
}
```
Limitations

Calling $fact$:

$$fact(4);$$
Limitations

Calling $fact$:

$$fact(4);$$

prints:

3 4
2 3
1 2
0 1
Limitations

Calling \texttt{fact}:

\texttt{fact(4);}

prints:

\begin{verbatim}
  3 4
  2 3
  1 2
  0 1
\end{verbatim}

We only allow access to current function and arguments
Related work

- Clements et al: Modeling an Algebraic Stepper (ESOP 2001)
Contributions

1. Defined semantics for aspects in a higher-order language
2. Explored consequences of these semantics
3. Developed lightweight implementation using continuation marks