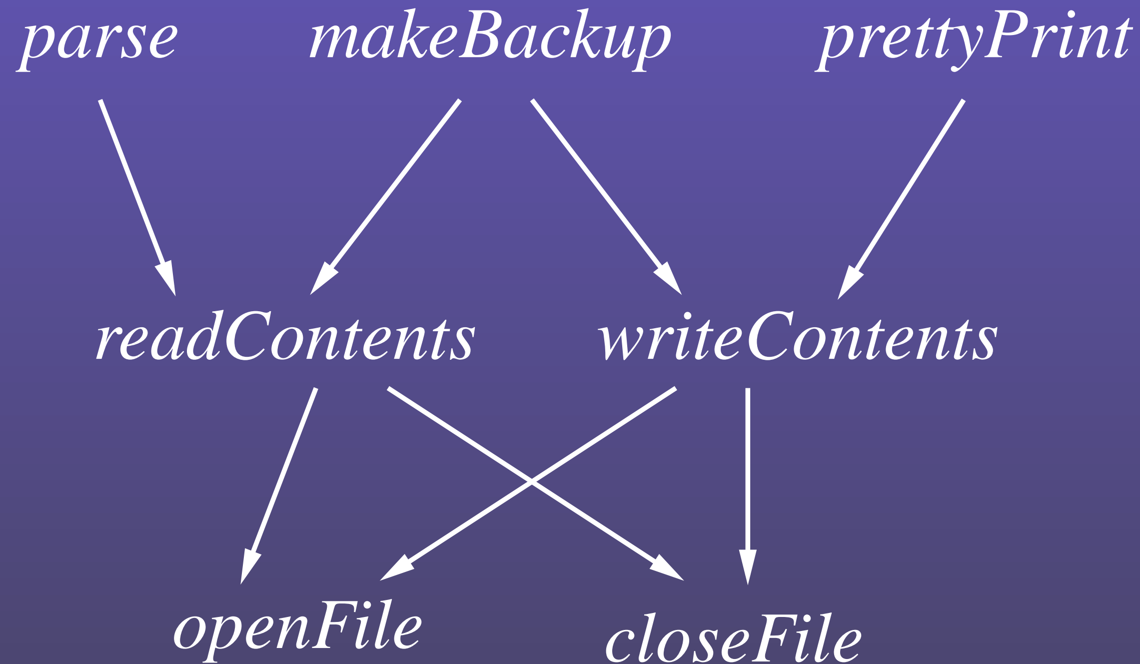
Pointcuts and Advice for Higher-Order Languages

David B. Tucker and Shriram Krishnamurthi

2003 March 20

An example



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:

call(**void** $closeFile$())

How to specify pcd's?

Calls to $closeFile$

In AspectJ:

call(**void** $closeFile$())

In our language:

$(\lambda\ (jpl)$

$\quad (eq?\ close\text{-}file\ (first\ jpl)))$

How to specify pcd's?

Calls to *closeFile* originating from *makeBackup*

In AspectJ:

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

How to specify pcd's?

Calls to *closeFile* originating from *makeBackup*

In AspectJ:

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

In our language:

(λ (*jpl*)
  (**and** (*eq? close-file* (*first jpl*))
      (*member make-backup* (*rest jpl*))))

How to specify pcd's?

$$(call\ f) \equiv (\lambda\ (jpl)\ (eq?\ f\ (first\ jpl)))$$

How to specify pcd's?

$$(call\ f) \equiv (\lambda\ (jpl)\ (eq?\ f\ (first\ jpl)))$$

$$(within\ f) \equiv (\lambda\ (jpl)\ (\textbf{and}\ (not\ (empty?\ (rest\ jpl)))$$
$$(eq?\ f\ (second\ jpl))))$$

How to specify pcd's?

$$(\mathit{call}\ f) \equiv (\lambda\ (\mathit{jpl})\ (\mathit{eq?}\ f\ (\mathit{first}\ \mathit{jpl})))$$

$$\begin{aligned}(\mathit{within}\ f) \equiv (\lambda\ (\mathit{jpl})\ (\textbf{and}\ &(\mathit{not}\ (\mathit{empty?}\ (\mathit{rest}\ \mathit{jpl})))\\ &(\mathit{eq?}\ f\ (\mathit{second}\ \mathit{jpl}))))\end{aligned}$$

$$\begin{aligned}(\&\&\ \mathit{pcd1}\ \mathit{pcd2}) \equiv (\lambda\ (\mathit{jpl})\ (\textbf{and}\ &(\mathit{pcd1}\ \mathit{jpl})\\ &(\mathit{pcd2}\ \mathit{jpl}))))\end{aligned}$$

How to specify pcd's?

$(\mathit{call}\ f) \equiv (\lambda\ (\mathit{jpl})\ (\mathit{eq?}\ f\ (\mathit{first}\ \mathit{jpl})))$

$(\mathit{within}\ f) \equiv (\lambda\ (\mathit{jpl})\ (\textbf{and}\ (\mathit{not}\ (\mathit{empty?}\ (\mathit{rest}\ \mathit{jpl})))\ (\mathit{eq?}\ f\ (\mathit{second}\ \mathit{jpl}))))$

$(\mathit{\&\&}\ \mathit{pcd1}\ \mathit{pcd2}) \equiv (\lambda\ (\mathit{jpl})\ (\textbf{and}\ (\mathit{pcd1}\ \mathit{jpl})\ (\mathit{pcd2}\ \mathit{jpl})))$

$$
\begin{aligned}
(\mathit{cflow}\ \mathit{pcd}) \equiv (\lambda\ (\mathit{jpl})\\
&(\textbf{cond}\\
&\quad [(\mathit{empty?}\ \mathit{jpl})\ \mathit{false}]\\
&\quad [\textbf{else}\ (\textbf{or}\ (\mathit{pcd}\ \mathit{jpl})\\
&\qquad\qquad ((\mathit{cflow}\ \mathit{pcd})\ (\mathit{rest}\ \mathit{jpl})))]))
\end{aligned}
$$

How to specify pcd's?

Rewrite examples as:

Calls to $closeFile$

$(call\ \ close\text{-}file)$

How to specify pcd's?

Rewrite examples as:

Calls to $closeFile$

$(\mathit{call}\ \ \mathit{close\text{-}file})$

Calls to $closeFile$ originating from $makeBackup$

$(\&\&\ (\mathit{call}\ \ \mathit{close\text{-}file})\ (\mathit{cflow}\ (\mathit{within}\ \ \mathit{make\text{-}backup})))$

How to specify pcd's?

Rewrite examples as:

Calls to *closeFile*

$(call\ close\text{-}file)$

Calls to *closeFile* originating from *makeBackup*

$(\&\&\ (call\ close\text{-}file)\ (cflow\ (within\ make\text{-}backup)))$

Showed how to define pcd's

Next: how to define advice

How to specify advice?

Procedure transformers:

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

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

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:

(**around** $Pcd\ Advice\ Body$)

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** ($call\ open\text{-}file$) $trace\text{-}advice$
    ($pretty\text{-}print\ input$ "file2")))

Review of scope in Java

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

Review of scope in Java

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

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

Review of scope in 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")
```

Review of scope in 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...?

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) $\Rightarrow$ "dave tucker"

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

Review of scope in Java

What happens here...?

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

Static scoping (Java) $\Rightarrow$ "dave tucker"

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

Dynamic scoping $\Rightarrow$ "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

((**around** (*call open-file*) *trace-advice*
  (λ (f) (*open-file* f)))
 "boston")

Also prints a trace message

Example #3

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

Example #3

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

This *does not* print a trace message

Example #3 revisited

Can we define aspects that *do* apply?

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

Example #3 revisited

Can we define aspects that *do* apply?

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

Example #3 revisited

Can we define aspects that *do* apply?

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

This *does* print a trace message

fluid-around aspects are *dynamically* scoped

- apply to join points during *evaluation* of body

Example #2 revisited

((**around** $(call\ open\text{-}file)$ $trace\text{-}advice$
  $(\lambda\ (f)\ (open\text{-}file\ f)))$
 "boston")

Example #2 revisited

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

Example #2 revisited

((**fluid-around** $(call\ open\text{-}file)$ $trace\text{-}advice$
$(\lambda\ (f)\ (open\text{-}file\ f)))$
"boston")

Does not print a message

Using dynamic aspects

Trace calls to $close\text{-}file$ that originate from $make\text{-}backup$

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*

```
(define (backup-system)
  (for-each make-backup
            (list "boston" "providence" "woonsocket")))

(fluid-around (&& (call close-file)
                  (cflow (within make-backup)))
              trace-advice
  (backup-system))
```

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:

```
(define protected-open-file
  (around (&& (call open-file)
              (! (until trusted? privileged?)))
          report-privilege-error
    (λ (f)
      (open-file f))))
```

Can export this function

Higher-order pointcuts

Since pointcuts are first-class, we could define *until*:

```
(define (until pcd1 pcd2)
  (λ (jpl)
    (cond
      [(empty? jpl) false]
      [else (or (pcd2 jpl)
                (and (pcd1 jpl)
                     ((until pcd1 pcd2) (rest jpl))))])))
```

Can you write this using **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:

```
(define (fact n)
  (w-c-m 'fact-arg n
    (if (zero? n)
        (begin (display (c-c-m 'fact-arg)) 1)
        (* n (fact (sub1 n))))))
```

Continuation marks example

$(fact\ 2)$

Continuation marks example

$(fact\ 2)$

```
⇒ (w-c-m 'fact-arg 2
     (* 2
       (w-c-m 'fact-arg 1
         (* 1
           (w-c-m 'fact-arg 0
             (begin (display (c-c-m 'fact-arg)) 1))))))
```

Continuation marks example

$(fact\ 2)$

$\Rightarrow$ (**w-c-m** 'fact-arg 2
    ($*$ 2
        (**w-c-m** 'fact-arg 1
            ($*$ 1
                (**w-c-m** 'fact-arg 0
                    (**begin** ($display$ (**c-c-m** 'fact-arg)) 1))))))

displays (0 1 2)

Implementation of dynamic aspects

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

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)

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:

```
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 $fact$:

$fact(4);$

prints:

3 4

2 3

1 2

0 1

We only allow access to current function and arguments

Related work

- Kiczales et al: An Overview of AspectJ (ECOOP 2001)
- Wand et al: A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming (FOAL 2002)
- Clements et al: Modeling an Algebraic Stepper (ESOP 2001)
- Orleans: Incremental Programming With Extensible Decisions (AOSD 2002)

Contributions

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