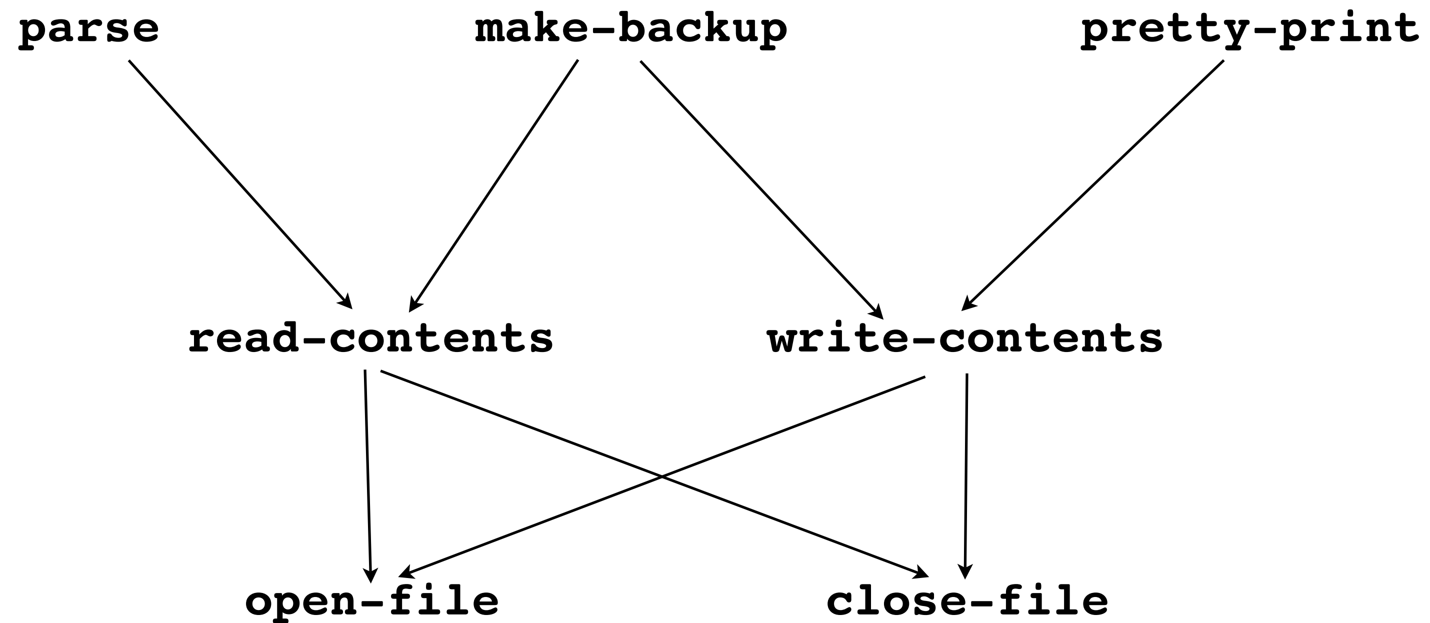
Higher-Order Aspects

AspectScheme

An Example



Possible Aspects

- trace calls to **`close-file`** originating from **`make-backup`**
- check for legal arguments to **`write-contents`**
- ensure the callee has permission to execute **`open-file`**
- *can we write these in a higher-order way?*

Why AOP in a H-O language

- many languages have higher-order, first-class functions
  - Scheme
  - ML
  - Haskell

Why AOP in a H-O language

- many languages have higher-order, first-class functions
  - Scheme
  - ML
  - Haskell

  - Perl
  - Python
  - Ruby

Why AOP in a H-O language

- many languages have higher-order, first-class functions
- what is the interaction between functional programming and aspect-oriented programming
  - simplify the specification of aspects?
  - define more general aspects?

Aside: Higher Order Functions

- an *accumulator* is a procedure that takes a number and adds it to its currently accumulated amount yielding the total

```
;;accumulator :: int → int
(define a (make-accumulator 0))
(define b (make-accumulator 100))
(a 10) ↦  10
(a 5)  ↦  15
(b 99) ↦ 199
(a 1)  ↦  16
(b 1)  ↦ 200
```

`make-accumulator` is *higher-order*

```
(define make-accumulator
  (lambda (acc)
    (lambda (n)                   ;new function
      (set! acc (+ acc n))
      acc)))
```

- a *higher-order* function constructs new functions
- `(define ((make-accumulator acc) n)`
  `(set! acc (+ acc n))`
  `acc)`

First-class Functions

- a common *shape* for operating on list data structures is
  1. **`cdr`**-ing down a list
  2. transforming each element
  3. returning another list of the new elements
- that shape is called **`map`**

```
(define (incr x) (+ x 1))
(map incr '(1 2 3 4 5)) ↦ (2 3 4 5 6)

(map string-length '("Hi" "Hola" "Bonjour")) ↦ (2 4 7)
```

`map` requires first-class functions

```
(define (map f l)
  (if (null? l)
      '()
      (cons (f (car l))
            (map f (cdr l)))))
```

- ***first-class functions can be arguments***
  - just like any other value

Another common shape

- a common *shape* for operating on list data structures is
  1. **`cdr`**-ing down a list
  2. transforming each element
  3. and combining the resulting element with the rest of the tranformed list
- that shape is called **`fold`**

```
(fold + 0 '(1 2 3 4 5)) ↦ 15

(map string-append "" '("hi" "hola" "bonjour"))
        ↦ "HiHolaBonjour"
```

H-O + F-C is powerful

```
(define (map f l)
  (fold (lambda (e l)
          (cons (f e) l))
        '()
        l))
```

Challenges

- How to specify aspects?
  - a F-C function may have 0, 1, or many names
  - second-class or first-class aspects?

Challenges

- How to specify aspects?
  - a F-C function may have 0, 1, or many names
  - second-class or first-class aspects?

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

How to Specify Aspects?

- two parts:
    - pointcut
    - advice

Aside: Aspects

- an aspect comprises two parts
  - a *pointcut*
    - identifies some collection of principled points
      - *join points*
    - in the execution of a program
  - an *advice*
    - alters the semantics of the join point
      - continue with different arguments
      - log information
      - decide not to continue at all

How to Specify Aspects?

- two parts:
    - pointcut
    - advice
  - will be first class
      - consistent with design of functional languages

- a pointcut
  - a predicate over a list of join points

- an advice
  - a join point transformer

Aside: Join Points

```
(define (incr x) (+ 1 x))

(incr 3)
```

- function call:

  ```
  (incr 3)
  ```

  – a pair of *target*: **`incr`** and *arguments*: **`'(3)`**
- function execution:

  ```
  (+ 1 3)
  ```

  – a pair of *target*: **`incr`** and *arguments*: **`'(3)`**

Join Point Stack

- offers access to calling context

jp jp jp jp jp jp

jp- | jp jp jp jp jp jp | jp+

How to specify Pointcuts

- calls to **`close-file`**

- AspectJ

```
call(void File.close())
```

How to specify Pointcuts

- calls to **`close-file`**

- AspectJ

```
call(void File.close())
```

- AspectScheme

```
(lambda (jp- jp jp+)
   (if (and (call-jp? jp)
            (eqv? (jp-target jp) close-file))
       '()
       #f))
```

Binding Pointcuts

- calls to **`close-file`** accessing the file

- AspectJ

```
call(void File.close(File)) && args(f)
```

- AspectScheme

```
(lambda (jp- jp jp+)
   (if (and (call-jp? jp)
            (eqv? (jp-target jp) close-file))
       (jp-args jp)
       #f))
```

How to specify Pointcuts

- calls to **`close-file`** originating from **`make-backup`**
- AspectJ

```
call(void File.close())
  && cflow(exec(void Backup.make()))
```

How to specify Pointcuts

- calls to **`close-file`** originating from **`make-backup`**
- AspectJ

```
call(void File.close())
  && cflow(exec(void Backup.make()))
```

- AspectScheme

```
(lambda (jp- jp jp+)
   (and (call-jp? jp)
        (eqv? (jp-target jp) close-file))
        (any (lambda (jp)
               (and (exec-jp? jp)
                  (eqv? (jp-target jp) make-backup)))
             jp+)))
```

Higher-Order Pointcuts

```
(define ((check type?) f) jp- jp jp+)
   (if (and (type? jp)
            (eqv? (jp-target jp) f))
       '()
       #f)))


(define (call f) ((check call?) f))

(define (exec f) ((check exec?) f))


(define (args) jp- jp jp+)
   (jp-args jp))
```

Pointcut Combinators

```
(define ((&& pc1 pc2) jp- jp jp+)
  (let ([v1* (pc1 jp- jp jp+)])
    (if v1*
        (let ([v2* (pc2 jp- jp jp+)]
          (if v2* (append v1* v2*) #f))
        #f)))

(define ((|| pc1 pc2) jp- jp jp+)
  (let ([v* (pc1 jp- jp jp+)])
     (if v* v* (pc2 jp- jp jp+))))

(define ((! pc) jp- jp jp+)
  (if (pc jp- jp jp+) #f '()))
```

Pointcut Combinators

```
(define ((cflow pc) jp- jp jp+)
  (let loop ([jp- jp-]
             [jp  jp ]
             [jp+ jp+])
    (if (null? jp+)
        #f
        (let ([v* (pc jp- jp jp+)])
          (if v*
              v*
              (loop (cons jp jp-)
                    (car jp)
                    (cdr jp+)))))))
```

How to specify Pointcuts

- calls to **`close-file`** originating from **`make-backup`** yielding the closing file and the backup file
- AspectJ

```
(call(void File.close(File)) && args(f)
   && cflow(exec(void Backup.make()) && args(b)))
```

- AspectScheme

```
(&& (call close-file)
    args
    (cflow (&& (exec make-backup)
               args)))
```

How to specify Advice

- calls to **`close-file`** originating from **`make-backup`** yielding the closing file and the backup file
- AspectJ

```
{ System.out.println("Backup " + b + " closing " + f);
  proceed(f, b); }
```

- AspectScheme

```
(lambda (proceed)
  (lambda (f b)
    (display `("Backup " ,b " closing " ,f))
    (proceed f b)))
```

- all advice is **`around`** advice

The `around` expression

- to install a pcd and advice, introduce

```
(around pcd adv
   body ...)
```

- for example

```
(let ([pcd (&& (call open-file) args)]
      [((adv p) f) (display `("Opening " ,f))
                   (p f))])
  (around pcd adv
     (open-file "Santiago")))
```

Aside: Lexical Scoping

```
(let ([x 1])
  (let ([(f y) (+ x y)])
    (let ([x 3])
      (f x))))
```

- lexical scoping yields ?
- dynamic scoping yields ?

Aside: Lexical Scoping

```
(let ([x 1])
  (let ([(f y) (+ x y)])
    (let ([x 3])
      (f x))))
```

- lexical scoping yields **4**
- dynamic scoping yields **6**

Scoping of `around`

- calls to **`close-file`** originating from **`make-backup`** yielding the closing file and the backup file
- AspectJ
  - all aspects are static and top-level
  - all aspects apply to that top-level scope

Scoping of `around`

- calls to **`close-file`** originating from **`make-backup`** yielding the closing file and the backup file
- AspectJ
  - all aspects are static and top-level
  - all aspects apply to that top-level scope
- AspectScheme
  - around aspects are ***statically scoped***
  - apply to all join points textually within that scope

Statically Scoped

```
(let ([pcd (&& (call open-file) args)]
      [((adv p) f) (display `("Opening " ,f))
                   (p f))])
  (around pcd adv
     (open-file "Santiago")))
```

?

Statically Scoped

```
(let ([pcd (&& (call open-file) args)]
      [((adv p) f) (display `("Opening " ,f))
                   (p f))])
  (around pcd adv
     (open-file "Santiago")))
```

```
Opening Santiago
```

Statically Scoped

```
(let ([pcd (&& (call open-file) args)]
      [((adv p) f) (display `("Opening " ,f))
                   (p f))])
  ((around pcd adv
     (lambda (f)
       (open-file f)))
   "Santiago")
```

?

Statically Scoped

```
(let ([pcd (&& (call open-file) args)]
      [((adv p) f) (display `("Opening " ,f))
                   (p f))])
  ((around pcd adv
     (lambda (f)
       (open-file f)))
   "Santiago")
```

```
Opening Santiago
```

Statically Scoped

```
(let ([(to-santiago f) (f "Santiago"))]
      [pcd (&& (call open-file) args)]
      [((adv p) f) (display `("Opening " ,f))
                   (p f))])
  (around pcd adv
    (to-santiago open-file)))
```

?

Statically Scoped

```
(let ([(to-santiago f) (f “Santiago”))]
      [pcd (&& (call open-file) args)]
      [((adv p) f) (display `(“Opening “ ,f))
                   (p f))])
  (around pcd adv
    (to-santiago open-file)))
```

No message!

- around aspects apply statically
  - only to operations lexicially in their scope
  - join points that occur textually in the aspect body

Dynamically scoped

```
(let ([(to-santiago f) (f "Santiago"))]
      [pcd (&& (call open-file) args)]
      [((adv p) f) (display `("Opening " ,f))
                   (p f))])
  (fluid-around pcd adv
    (to-santiago open-file)))
```

?

Dynamically scoped

```
(let ([(to-santiago f) (f "Santiago"))]
      [pcd (&& (call open-file) args)]
      [((adv p) f) (display `("Opening " ,f))
                   (p f))])
  (fluid-around pcd adv
    (to-santiago open-file)))
```

```
Opening Santiago
```

- fluid-around aspects apply dynamically
  - only to operations dynamically in their scope
  - join points that occur during the evaluation of the body

Dynamically Scoped

```
(let ([pcd (&& (call open-file) args)]
      [((adv p) f) (display `("Opening " ,f))
                   (p f))])
  ((fluid-around pcd adv
     (lambda (f)
       (open-file f)))
   "Santiago")
```

No message!

- the body of the **`fluid-around`** has completed before the anonymous function is applied

Using Static Aspects

- Ensure callee has permission to **`open-file`**
- Use stack inspection:
  - only trusted calls until permission granted

```
(define protected-open-file
   (let ([pcd (&& (call open-file)
                  (! (until trusted? priviledged?)))]
         [adv report-priviledge-error])
     (lambda (f)
       (open-file f))))
```

- export **`protected-open-file`** instead of **`open-file`**

Higher-Order Advice

```
(before pcd adv
  body ...)
```

- Want to ensure `proceed` called
  - exactly once
  - with original arguments
- – this is extensional advice only

```
(let ([(make-before-adv adv) proceed) args)
          (let ([(new-proceed ignored-args)
                      (error 'as "proceeding in before")])
            (begin (adv new-proceed args)
                      (proceed args)))])
  (around pcd (make-before-adv adv)
    body ...))
```

Summary

- extensible pointcuts and advice language
  - higher-order and first-class functions
    - allow us to easily write our own pointcuts
    - allow us to customize advice behaviours
- two new kinds of scoping for aspects
  - lexical (static)
    - properties paralleling the program lexical structure
    - propagate into higher-order procedures
      - and are carried with them
  - dynamic (fluid)
    - properties paralleling the program dynamic structure
    - propagate along the call structure

Implementation

Techniques ... not real code

Requirements

- join points: access the call-stack
  - fluid-let
  - continuation marks
- around: new syntax
  - hygenic macros
- weaving: intercept lambda and application
  - lambda is easy: hygienic macro
  - application: reader macros
    - PLT supplied it automatically, so just hygienic macros

Continuation Marks

```
(with-continuation-mark tag value
  body ...)

(get-continuation-marks)



(define (jp-context) (get-continuation-marks 'JP))

(define-syntax with-jp
  (syntax-rules ()
    [(_ jp body ...) (with-continuation-mark 'JP jp
                        body ...)]))
```

Hygienic Macros

```
(define-syntax succeed
  (syntax-rules ()
    [(_ exp) (if exp '() #f)]))


(define ((check type?) f) jp- jp jp+)
   (succeed (and (type? jp)
                 (eqv? (jp-target jp) f))))

(define ((! pc) jp- jp jp+)
   (succeed (not (pc jp- jp jp+))))
```

Aspect Scoping

```
(define static-aspects  '())
(define dynamic-aspects '())

(define (current-aspects)
   (append static-aspects dynamic-aspects))

(define-syntax lambda/static
  (syntax-rules ()
      [(_ (arg ...) body ...)
       (let ([aspects (static-aspects)])
         (lambda (arg ...)
           (fluid-let ([static-aspects aspects])
             body ...))))]))
```

Weaving

```
(define-syntax app/weave
  (syntax-rules ()
      [(_ f a ...) (app/weave/rt f a ...)]))

(define (weave fun-val jp- jp jp+ aspects)
  (fold (lambda (aspect fun)
          (cond
             [((aspect-pc aspect) jp- jp jp+)
              => ((aspect-adv aspect) fun)]
             [else fun]))
        fun-val
        aspects))
```

Weaving

```
(define (app/weave/rt fun arg ...)
  (if (primitive? fun)
      (apply fun args)
      (let ([jp (make-call-jp fun (list arg ...))]
            [jp+ (jp-context)])
        (with-jp jp
          ((weave
             (lambda (arg ...)
               (with-jp (make-exec-jp fun (list arg ...))
                  (fun arg ...)))
             '()
             jp
             jp+
             (current-aspects))
           arg-vals)))))
```

fluid-around and around

```
(define-syntax fluid-around)
  (syntax-rules ()
    [(_ pc adv body)
     (fluid-let ([dynamic-aspects
                  (cons (make-aspect pc adv))])
        body ...)]))

(define-syntax around)
  (syntax-rules ()
    [(_ pc adv body)
     (fluid-let ([static-aspects
                  (cons (make-aspect pc adv))])
        body ...)]))
```

Language-Defining Macros

```
(provide (rename [lambda/static lambda]
                 [app/weave     #%app]))
```

Questions?