

# ***Specializing Continuations***

***a Model for Dynamic Join Points***

**Christopher Dutchyn**

**University of Saskatchewan**



**1**

## Q: What is an Aspect?

- A: give examples
  - Distribution / tracing / instrumentation / ...



## Q: What is an Aspect?

- A: give examples
  - Distribution / tracing / instrumentation / ...
- A: give implementations
  - It's what AspectJ (and any number of others) do



## Q: What is an Aspect?

- A: give examples
  - Distribution / tracing / instrumentation / ...
- A: give implementations
  - It's what AspectJ (and any number of others) do
- ... lead to poor insight regarding
  - *what aspects are good for?*
  - *how to best use them?*



# The key is *Modularity*

**Q: What do aspects modularize?**



# The key is *Modularity*

**Q: What do aspects modularize?**

**A: cross-cutting concerns.**



# The key is *Modularity*

**Q: What do aspects modularize?**

**A: cross-cutting concerns.**

**?!**



# Kinds of Aspect Orientation

- Static aspects
  - Open classes
- Composition filters
- Object graph traversal (*Demeter*)
- Dynamic join points, pointcuts, and advice
  
- **Space is too large for a coherent answer**



# Modeling Dynamic Aspects

- Join points
  - “*principled points in the execution*”
- Pointcuts
  - “*a means of identifying join points*”
- Advice
  - “*a means of affecting the semantics at those join points*”



# Model Development

Big-step semantics

$$\frac{\Gamma \vdash f \Downarrow \lambda x. b \quad \Gamma \vdash a \Downarrow v \quad \Gamma[x \mapsto v] \vdash b \Downarrow w}{\Gamma \vdash (f a) \Downarrow w}$$

introduce continuations

*Wand; Danvy; and many others*

defunctionalize continuations

Small-step semantics

$$\begin{aligned} \langle\langle e, \rho, \langle \text{start} \rangle \rangle\rangle &\Rightarrow \langle\langle e, \rho, \langle \text{halt} \rangle \rangle\rangle \\ \langle\langle (f a), \rho, \kappa \rangle\rangle &\Rightarrow \langle\langle a, \rho, \langle \text{call } f, \rho, \kappa \rangle \rangle\rangle \\ \langle\langle v, \rho, \langle \text{call } f, \rho, \kappa \rangle \rangle\rangle &\Rightarrow \langle\langle f, \rho, \langle \text{exec } v, \rho, \kappa \rangle \rangle\rangle \\ \langle\langle \lambda x. b, \rho, \langle \text{exec } v, \rho, \kappa \rangle \rangle\rangle &\Rightarrow \langle\langle b, [x \mapsto v] \oplus \rho, \kappa \rangle\rangle \\ \langle\langle v, \rho, \langle \text{halt} \rangle \rangle\rangle &\Rightarrow v \end{aligned}$$



# Example: PROC Language

- Functions
  - 1st order, 2nd class
- Globals with mutation
- Standard syntax elements
  - `if`
  - Application
  - Primitives



# Continuation Frames

## Auxiliary

- facilitate eval regime
  - eager vs lazy
- `testF -- if`
- `randF -- args`
- `konsF -- args`
- `rhsF -- set`

## Non-auxiliary

- Carry essential semantics of language
- `getF`
- `setF`
- `callF`
- `execF`

*“nonlogical symbols”*



# Pointcuts -- identify frames

- **callC**
  - convert a procedure name to a procedure value
    - NB: accepts an internal value: an identifier
  - then continue to execF
- **execC**
  - accept arguments and execute procedure
- **getC**
  - accept global location and provide its value
- **setC**
  - accept global location and update its value



# Pointcuts - combinators

- **and**
- **or**
- **not**



# Aspect weaving is *dispatch*

```
(define ((adv-step advs) f k) v)
;:adv* → (frm × cont) → !val
  (let loop ([advs advs])
    (cond [(null? advs) ((base-step f k) v)]
          [(match-pc (caar advs) v f) =>
            (lambda (m)
              (eval (cdar advs)
                    (extend-env `(%proceed
                                %advs .
                                , (match-ids m)
                                ` (, (match-proc m)
                                , (cdr advs) .
                                , (match-vals m))
                                empty-env)
                    k))])])
```



# Predictive Power of Model

- Our account requires a new join point
  - for advice execution:
    - **advF**
- Kiczales intuited that such a thing is required
  - **adviceExecution**
- 



## Wrinkle: `cflowbelow` pointcut

- identifies join points based on control-flow context
- tail-call optimization discards context
- recovering context
  - 1) keep all of it
  - 2) preserve needed structure [CC'03]
    - dynamically threaded stack data structure
    - **or** state effect



## **cflowabove pointcut**

- Adds to ability to bound the context search from above
- **within**
  - Exclude subordinate procedure calls
- **enclosingexecution**
  - Stop at the next higher calling scope
- Not strictly necessary, but expressive



# Fundamental Construction

- continuations arise naturally in big-step to small-step translation
- frames arise mechanically in defunctionalization of continuations
- **no new language construct required**
  - no continuation marks [Dutchyn, Tucker, Krishnamurthi]
  - no context labels [Dantas, Walker, Washburn, Weirich]
  - no rewrite points [Aßmann, Ludwig]
  - no awkward thunks [Wand, Kiczales, Dutchyn]
  - no predicate dispatch [Orleans]



no continuation marks [Dutchyn, Tucker, Krishnamurthi]

no context labels [Dantas, Walker, Washburn, Weirich]

no rewrite points [Aßmann, Ludwig]

no awkward thunks [Wand, Kiczales, Dutchyn]

no predicate dispatch [Orleans]

# AOP and Reflection?

- “AOP offers tamed reflection: the significant power without the shooting yourself in the foot”

Jim Huginin, MS PDC'05



# Why Reflection?

- An implementation: reflect and reify
  - and Filinski (various)
  - Mendhekar & Friedman (1996)

```
(define star
  (lambda (k x)
    (if (run-time-optimizable? x)
        (k (down-arrow (f x)))
        (k (down-arrow x)))))
```

- reify the continuation
- examine the continuation with predicates
- reflect with new value and new continuation



# Why Reflection?

- An implementation: reflect and reify
  - and Filinski (various)
  - Mendhekar & Friedman (1996)

```
(define kstar
  (lambda (k v)
    (if (pointcut-matches? k)
        ((down-arrow (advise k)) v)
        ((down-arrow k) v))))
```

```
(define star
  (lambda (k x)
    (if (run-time-optimizable? x)
        (k (down-arrow (f x)))
        (k (down-arrow x)))))
```

- reify the continuation
- examine the continuation with predicates
- reflect with new value and new continuation



## Example: OBJECT Language

- Classes with single inheritance
- Methods
  - 1st order, 2nd class
- Single constructor
- Fields with mutation



# Pointcuts *discovered* in Semantics

- `callC`
- `execC`
- `getC`
- `setC`
- `initC`
- `allocC`
  
- `super --`
- `superInit --`

Frame Activation	Pointcut	AspectJ
$(field_{location} i) \triangleright (getfield_{frame} o)$	<code>getfield o.i</code>	<code>getfield o.i</code>
$o \triangleright (setfield_{frame} field_{location} i)$	<code>setfield o i</code>	<code>setfield o.i</code>
$v^* \triangleright (dispatch_{frame} o i)$	<code>dispatch o.i(...)</code>	<code>call o.i(...)</code>
$(method_{location} i) \triangleright (exec_{frame} o v^*)$	<code>exec o.i(...)</code>	<code>exec o.i(...)</code>
$v^* \triangleright (allocate_{frame} i)$	<code>alloc i(...)</code>	<code>init i(...)</code>
$(class i) \triangleright (init_{frame} v^*)$	<code>init i(...)</code>	<code>preinitialize i(...)</code>

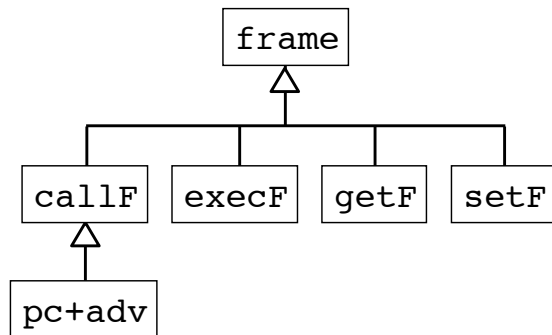
Figure 51: Object-Oriented Dynamic Join Points



# Pointcuts and advice **specialize control**

```
(pointcut (fooCalls (callC foo))  
(pointcut (barCalls (callC bar))  
  
(around fooCalls || barCalls  
  (display "foo called")  
  (proceed x))
```

- describes a specialized kind of **callF** frame, with specialized behaviour



# Object/Aspect Duality: Auburn

## Object

*data structure with associated operations*

- operations are dispatched
  - classes specialize operation to execute
- fields hold values
- operations are overridden by sub-methods
  - **super**

## Aspect

*control structure with associated action*

- construction is dispatched
  - advice specializes the frame to construct
- fields hold context
- action is overridden by advice
  - **proceed**



# Static Aspects

- elaboration :: declaration  $\longrightarrow$  class
  - it's another interpreter



- it has a big-step semantics
  - which yields a small-step semantics
  - which identifies continuation frames/join points
  - which can be advised
    - class construction allows inter-type declarations
    - method population yields type replacement



## Future Directions: Aspect Types

- *abstract control types*
  - more than just input/output value typing
  - needs to capture types of delimited continuations
    - polarized logic from Shan et al.
- type checking is (abstract) interpretation
  - what are the join points?
  - what can advising type checks do?



## Take Home Messages

- JP&A AOP is not preprocessor technology
  - don't confuse implementation of X with X
    - e.g. objects are not virtual function tables
  - there may be other implementations and insights
- JP&A AOP  $\approx$  modularization of continuations
  - parallels object  $\approx$  modularization of data
  - this has interesting implications for
    - Ruby
    - Python
    - Scalaand other OO languages that intend to support continuations



no continuation marks [Dutchyn, Tucker, Krishnamurthi]

no context labels [Dantas, Walker, Washburn, Weirich]

no rewrite points [Aßmann, Ludwig]

no awkward thunks [Wand, Kiczales, Dutchyn]

no predicate dispatch [Orleans]

## Finally,

- simple, elegant, evocative, general concepts
- applied with the rigour of a mathematician

Thank you, Mitch



no continuation marks [Dutchyn, Tucker, Krishnamurthi]

no context labels [Dantas, Walker, Washburn, Weirich]

no rewrite points [Aßmann, Ludwig]

no awkward thunks [Wand, Kiczales, Dutchyn]

no predicate dispatch [Orleans]

# Discussion

