Specializing Continuations

a Model for Dynamic Join Points

Christopher Dutchyn
University of Saskatchewan
actually: What is an Aspect?

- Give examples
  - Distribution / tracing / instrumentation / …

- 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

• So the question is

What do aspects modularize?
The key is Modularity

So the question is

What do aspects modularize?
In general: crosscutting concerns

- 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”
Two Interacting Abstractions: Join point and Advice

[p proc(x) (if (call = 0 x) (raise zero) 1)]

(beg\in (call p 1) (call p 2))

[a advise (exec p v) (try (proceed v) (catch zero ...))]
Third Abstraction: Pointcut

[p proc(x) (if (call = 0 x) (raise zero) 1)]

(begin (call p 1) (call p 2))

Pointcut

[a advise (exec p v) (try (proceed v) (catch zero ...))]
Interaction Between Pointcut and Advice

[p proc(x) (if (call = 0 x)
  (raise zero)
  1)]
(bEGIN (call p 1)
  (call p 2))
**Idea**

- A model of
  - dynamic join points,
  - pointcuts,
  - and advice,
    based on a continuation-passing style interpreter,
- provides a fundamental account of these AOP mechanisms.
Without Continuations

(define (f x)
  do-stuff
  (g x)
  do-more-stuff)

(define (g x) ...)

do-stuff

call g

...
Continuations

[Strachey’67, Landin’68,...]

```
(define (f x)
  do-stuff
  (g x)
  do-more-stuff)
```

```
(define (g x) ...)
```

```
  (define (g x) ...)
```

```
  do-more-stuff
```

```
  do-stuff
```

```
  call g
```

```
  ... 
```
Model Development

- Begin with big-step semantics
  - definition of values, expressions
  - semantic definition of `eval`

- Apply CPS transformation
  - yields continuations (as lambdas)
  - generates definition of `apply`

- Defunctionalize
  - yields identifiable frames in continuation structure
Defunctionalization  [Reynolds ’98, Ager+ ’03]

- Procedures have structure
  - identifiers (argument names)
  - environment
  - expression (machine code)

- Continuations as escape procedures
  - have simple list/tree structure
    - fixed identifiers (next-continuation, argument)
    - predetermined environment
    - given semantics involving one operation
PROC Language

- Functions
  - 1st order, 2nd class

- Globals

- 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
Insight ... Principle

Insight: frames align with dynamic join points

Principle:

A dynamic join point is modeled as a state in the interpreter where values are applied to non-auxiliary continuation frames.
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
Matching

- Take a pointcut, value and frame
- Capture
  - necessary context values
- Yields function to replace frame and value
  - Bind in a user-parameterized reflective monad
    - Mendhekar and Friedman
(define (match-pc c v f)
  ;; (pcut × val × frm) → match
  (cond ;;...other cases omitted
    [(and (callC? c) (callF? f)
            (eq? (callC-pid c) (callF-id f)))
      (make-match (callC-ids c) v
                  (lambda (nv)
                    (values nv f)))]
    [(and (execC? c) (execF? f)
            (eq? (lookup-proc (execC-pid c)) v))
      (make-match (execC-ids c)
                  (execF-args f)
                  (lambda (nv)
                    (values v (make-execF nv))))])}
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
Weaving is dispatch

\[
\text{define } ((\text{adv-step } \text{advs} \ f \ k) \ v) \\
\text{let } \text{loop } ([\text{advs} \ \text{advs}]) \\
\quad \text{cond } [(\text{null? } \text{advs}) ((\text{base-step } \ f \ k) \ v)] \\
\quad [(\text{match-pc} (\text{caar } \text{advs}) \ v \ f) => \\
\quad \quad \text{lambda } (m) \\
\quad \quad \text{eval } (\text{cdar } \text{advs}) \\
\quad \quad \text{(extend-env } '(\%proceed \\
\quad \quad \quad \%advs . \\
\quad \quad \quad , (\text{match-ids } m)) \\
\quad \quad \quad ',(\text{match-prcd } m) \\
\quad \quad \quad , (\text{cdr } \text{advs}) . \\
\quad \quad \quad , (\text{match-vals } m)) \\
\quad \quad \text{empty-env}) \\
\quad \quad k))] \\
\quad [\text{else } \text{loop } ((\text{cdr } \text{advs}))]]])
\]
Model Accounts for Observation

• Our account requires a new join point
  – We needed a new continuation frame
    • advF

• Arises naturally in the model
  – Rather than adding (without explanation)
    • AspectJ
    • And others
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]
Dynamic Semantic Model

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Model Element</th>
<th>Interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>join point</td>
<td>frame activation</td>
<td>dispatch</td>
</tr>
<tr>
<td>advice</td>
<td>behaviour specification</td>
<td>dispatch</td>
</tr>
<tr>
<td>pointcut</td>
<td>frame identifier</td>
<td>dispatch table</td>
</tr>
</tbody>
</table>

- Distills other descriptions to essentials
  - continuation marks
  - context labels
  - thunks

- Key insight: dynamic join points, pointcuts and advice
  - provide mechanism to modularize and specialize control structure
Elegant, Evocative Model

- based on a fundamental language construct
- pointcuts align well with existing AOP languages
  - adds `cflowabove` for simpler coding
  - explains provenience of `adviceexecution`
- clarifies relationship of DJP and reflection
- framework for understanding that dynamic aspects modularize control structure
Future Directions

• Object - Aspect Duality
  – Dynamic aspects modularize control (and associated operations)
    • Just as object modularize data (and associated operations)

<table>
<thead>
<tr>
<th>Frame Activation</th>
<th>Pointcut</th>
<th>AspectJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>(field_location i) ➔  (getfield_frame o)</td>
<td>getfield.o.i</td>
<td>getfield.o.i</td>
</tr>
<tr>
<td>o ➔ (setfield_frame field_location i)</td>
<td>setfield.o.i</td>
<td>setfield.o.i</td>
</tr>
<tr>
<td>v* ➔ (dispatch_frame o i)</td>
<td>dispatch.o.i(...)</td>
<td>call.o.i(...)</td>
</tr>
<tr>
<td>(method_location i) ➔ (exec_frame o v*)</td>
<td>exec.o.i(...)</td>
<td>exec.o.i(...)</td>
</tr>
<tr>
<td>v* ➔ (allocate_frame i)</td>
<td>alloc.i(...)</td>
<td>init.i(...)</td>
</tr>
<tr>
<td>(class i) ➔ (init_frame v*)</td>
<td>init.i(...)</td>
<td>preinitialize.i(...)</td>
</tr>
</tbody>
</table>

Figure 51: Object-Oriented Dynamic Join Points

• Category theory?
Future Directions

• Reflective Monads
  – Within the continuation monad
    • identify and operate on the continuation and value
  – à la Mendhekar & Friedman and Filinski

– Lost “chapter 3a” of my dissertation
Future Directions

• Typing Aspects -- *abstract control types*
  – Value typing (mundane PE) isn’t enough
    • Must abstract the control restructuring too
  – Rinard et al., Katz et al., and others

• Second half of my dissertation
  – But, more sophisticated
    • Take polarized logic from Shan
    • And effect typing from many others
Future Directions

• Static Aspects
  – Introduce an account of phase separation
    • Elaboration vs. execution
  – Continuations in elaboration
    = static join points?

  – Masuhara and Kiczales (ECOOP 2003)
Discussion