

Effects for Specialized Continuations

Aspects Provide Layered Effects

Christopher Dutchyn

University of Saskatchewan



Organization

- My model for dynamic JP&A aspects
 - “specializing continuations”
- Reformulate as a restricted form of monadic reflection
- Discuss the effect layering that this yields
 - identify four aspects that need effect typing



My Goal: What is an Aspect?

- Others
 - Give examples
 - Distribution / tracing / instrumentation / ...
 - Give implementations
 - It's what AspectJ (and any number of others) do
- all leading 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?



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)]
```

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

```
[a advise (exec p v)
          (try (proceed v)
              (catch zero ...))]
```

**Join
point**

Advice



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 ...))]
```

**Join
point**

Advice



Interaction Between Pointcut and Advice

```
[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 ...))]
```

Join
point

Advice



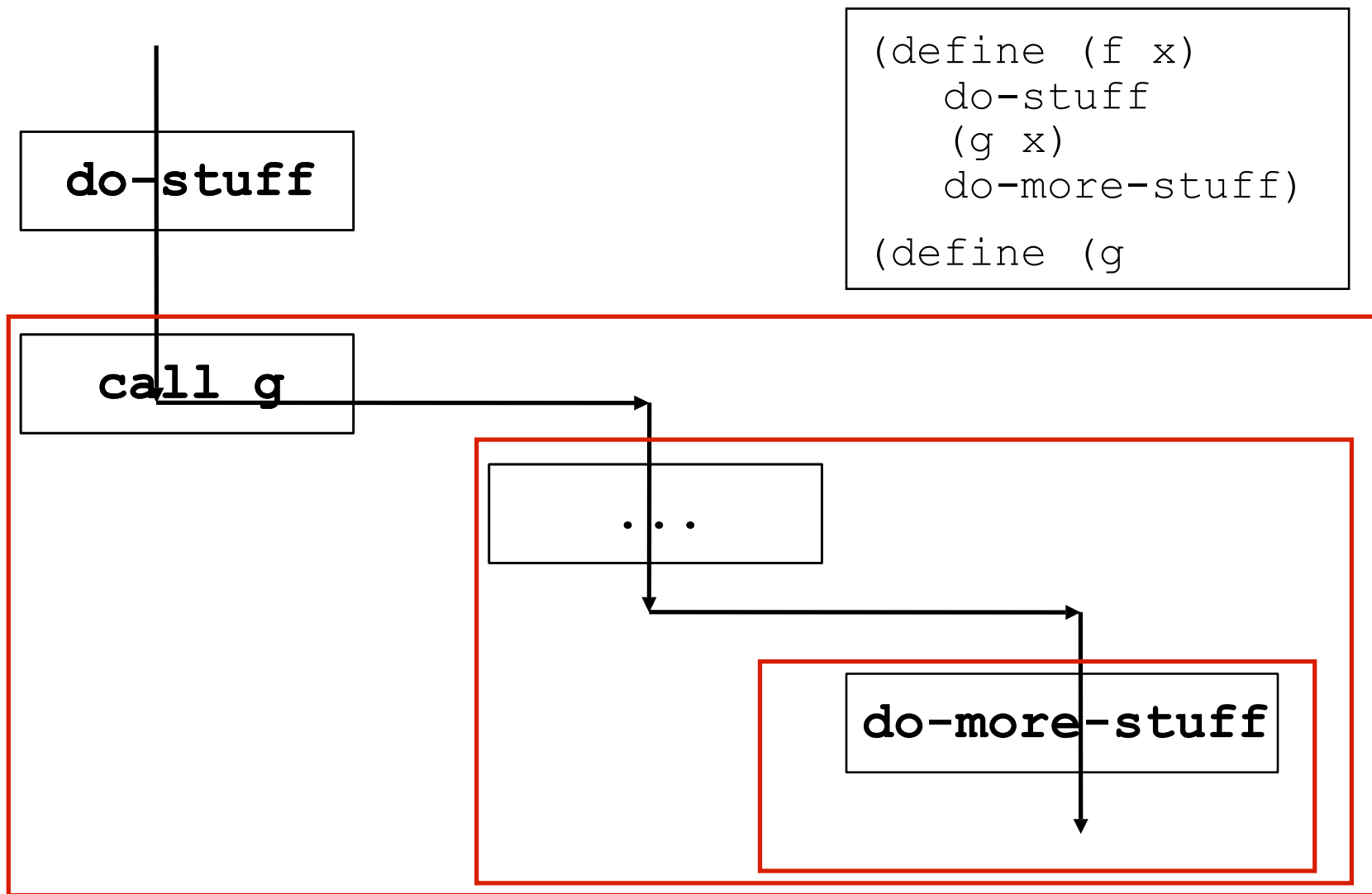
Idea

- A model of
 - dynamic join points,
 - pointcuts,
 - and advice,based on a continuation-passing style interpreter,
- provides a fundamental account of these AOP mechanisms.

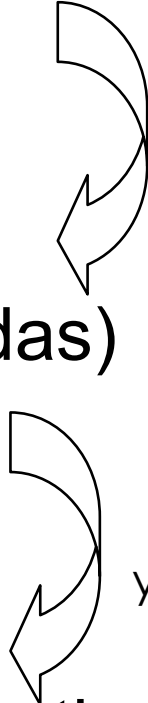


Continuations

[Strachey'67, Landin'68,...]



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
- introduces auxiliary continuations
- yields frame structures
- 



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

Dynamic Join Point	
Value Consumed	Frame Information
(loc i _{global})	▶ (getF)
(loc i _{global})	▶ (setF val)
(val*)	▶ (callF i _{proc})
(proc i _{proc})	▶ (execF val*)

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
 - pointcut,
 - value
 - continuation frame
- Capture
 - necessary context values
- Yields function to replace frame and value
 - Bind in a user-parameterized *reflective monad*
 - *Mendhekar and Friedman[1996], Filinski[1997]*



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-prcd m)
            , (cdr advs) .
            , (match-vals m))
            empty-env))
```



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



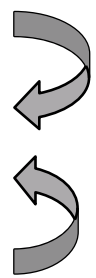
Fundamental Construction

- 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

Abstraction	Model Element	Interaction
join point	frame activation	dispatch
advice	behaviour specification	dispatch table
pointcut	frame identifier	



- 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



II: Monadic Reflection

- *reset*
 - places “marker” into the continuation frame list (*tree*)
- *shift*
 - gives the block of frames to nearest reset as procedure and allows it to be replaced
- allows implementation of
 - *reify* -- the state of computation
 - *reflect* -- a new state of computation



AOP is Monadic Reflection

- A restricted form
 - we place a reset marker after each frame
 - we shift before every apply, using
 - pre-programmed hierarchy of replacement frames



Monadic Reflection is Powerful

- Allows us to control effects
 - to layer them
 - by “Representing Layered Monads”
- Filinski 1997 -- modular ways of expressing
 - exceptions $V \dashrightarrow V + 1$
 - state $V \dashrightarrow (V, S)$
 - concurrency $V \dashrightarrow [V]$
- And he could combine them too



III Effect Typing

- this is the work-in-progress step
- I have four general aspects
 - state over exceptions -- software transactional memory
 - concurrency control
 - exceptions across concurrency
 - resumable exceptions
- want a lightweight annotations (*effect types*)
- expressing programmer intention
- for effect combinations generated by aspects



Transactional Memory

- // Transaction -- one implementation
- aspect MakeTransactional
- // need effect annotation to describe region involved
- // need “holes” -- logging, etc.
- {
 - void cacheState() { ... }
 - void restoreState() { ... }
 - around (COMPUTATION) {
 - cacheState();
 - try {
 - » proceed();
 - } catch Exception e {
 - » restoreState();
 - }
 - }



Transactional Memory

- Way to describe transactional region
- Way to describe acceptable “non-transactional” state/IO behaviour
 - log message that transaction aborted
- Provide validation of user intention
 - including ensuring no other code treats the transactional memory in a non-transactional way



Concurrency

- // Concurrency -- one implementation
- aspect MakeAsynchronous {
 - static public Thread t;
 - around (COMPUTATION) {
 - t = Thread.new(new Runnable() {
 - » public void run() { proceed(); } });
 - t.run;
 - } }
- aspect SynchronousBarrier {
 - around (ANOTHER_COMPUTATION) {
 - proceed();
 - if (NULL != MakeAsynchronous.t) { wait(t);
 - }
- }



Concurrency

- How to write down the intended
 - synchronous
 - asynchronous
 - barriers?

- Should we inform the user if two asynchronous tasks will interact over state
 - both write to same variables?
 - one reads what other writes?



Exceptions and Concurrency



Exceptions and Concurrency

- // Propagate exceptions across computations
- aspect PropagateExceptions {
 - around (COMPUTATION) {
 - static Exception except = NULL;
 - try { proceed(); }
 - catch Exception e {
 - » except = e;
 - » throw(e);
 - }
 - around (cflowbelow(ANOTHERCOMPUTATION)) {
 - if (NULL != except) { throw e }
 - }
- }



Exceptions Across Concurrency

- Lightweight annotation saying
 - how exceptions should propagate across
 - from where?
 - to where -- all? some?
 - both directions?
 - what points in the other computation
 - we've said every one -- a truly unusual pointcut



Resumable Exceptions



Resumable Exceptions

- // Retry a computation in event of exception
- aspect ResumableExceptions {
 - around (cflowbelow(COMPUTATION)) {
 - try {
 - » proceed();
 - } catch Exception e {
 - » fix_value();
 - » proceed();
 - }



Resumable Exceptions

- Which exceptions?
- Everywhere in the computation?
- Is the repair dependent on the location it occurs?



Summary

- Aspects provide (much of) the power of monadic reflection to **control and layer effects**
- I believe that's their raison d'être, not
 - fixing incomplete languages
 - open classes,
 - multiple dispatch,
 - multiple inheritance
 - making a module out of a bug fix



Summary

- But, without some effect type annotations
 - and associated checking
- Aspects won't develop in this way
- Put another way:
 - without a contract specifying the programmers' intentions, there are two poles:
 - every effect is predetermined by language semantics
 - every aspect that changes behaviour is wrong ...
 - no effect is predetermined
 - every aspect is correct
 - programmer used wrong one without notice



Aspects **NEED** effect types

- at least as much as any other construct
 - procedures, methods, ...
- and probably more!



Future Directions

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

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

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
 - á 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

