Specializing Continuations
A Model for Dynamic Join Points

Christopher J. Dutchyn
Computer Science, University of Saskatchewan
dutchyn@cs.usask.ca

ABSTRACT

By modeling dynamic join points, pointcuts, and advice in a defunctionalized continuation-passing style interpreter, we provide a fundamental account of these AOP mechanisms. Dynamic join points develop in a principled and natural way as activations of continuation frames. Pointcuts arise directly in the semantic specification as predicates identifying continuation frames. Advice models procedures operating on continuations, specializing the behaviour of continuation frames. In this way, an essential form of AOP is seen, neither as meta-programming nor as an ad hoc extension, but as an intrinsic feature of programming languages.

1. INTRODUCTION

Current programming languages offer many ways of organizing code into conceptual blocks, through functions, objects, modules, or some other mechanism. However, programmers often encounter features that do not correspond well to these units of organization. Such features are said to *scatter* and *tangle* with the design of a system, because the code that implements the feature appears across many program units. This scattering and tangling may derive from poor modularization of the implementation; for example, as a result of maintaining pre-existing code. But, recent work[Coady et al., 2004, De Win et al., 2004, Spinczyk and Lohmann, 2004] shows that, in some cases, traditional modularity constructs cannot localize a feature's implementation. In these cases, the implementation contains features which inherently *crosscut* each other.[1] In a procedural language, such a feature might be implemented as parts of disjoint procedures; in an object-oriented language, the feature might span several methods or classes.

[1]Strictly speaking, crosscutting is a three-place relation: we say that two concerns crosscut each other with respect to a mutual representation. The less rigorous 'two concerns crosscut each other' means that they crosscut each other with respect to an implementation that closely parallels typical executable code. Traditional modularity constructs, such as procedures and classes, have a close parallel between source and executable code.

Sixth International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007) March 13, 2007, Vancouver, BC, Canada.
Copyright 2007 ACM ISBN 1-59593-671-4/07/03 ...$5.00.

These crosscutting features inhibit software development in several ways. For one, it is difficult for the programmer to reason about how the disparate pieces of the feature interact. In addition, they compound development workload because features cannot be tested in isolation. Also, they prevent modular assembly: the programmer cannot simply add or delete these features from a program, since they are not separable units. Aspect-oriented programming (AOP) is intended to provide alternative forms of modularity, to extract these crosscutting features into their own modules. As a result, the code more closely resembles the design. AOP subsumes a number of different modularity technologies, some pre-existing, such as open classes and rewriting systems, and some more unconventional, including dynamic join points and advice. This work provides a novel semantic description of this latter system of dynamic join points, pointcuts, and advice. From this semantics, we provide a new viewpoint to what this form of AOP can modularize well, and eliminate the ad hoc foundation for dynamic join points, pointcuts, and advice.

By modeling dynamic join points, pointcuts, and advice in a defunctionalized continuation-passing style interpreter, we provide a fundamental account of these AOP mechanisms. Dynamic join points no longer rely on intuition to provide "well-defined points in the execution of a program"[Kiczales et al., 2001], but arise in the language semantics in a principled and natural way as activations of continuation frames. Pointcuts arise directly in the semantic specification as predicates identifying continuation frames. Advice models procedures operating on continuations, the dual of its usual behaviour as value transformers. Advice is shown as specializing the behaviour of continuation frames, leading us to understand dynamic join points, pointcuts, and advice as enabling the modularization of control in programs. In this way, an essential form of AOP is seen, neither as meta-programming nor as ad hoc extension, but as an intrinsic feature of programming languages.

We begin our presentation by giving direct semantics for an idealized procedural language, in Section 2. We transform to the continuation passing semantics in Section 3, and identify the three model elements within that semantics in Section 4. Following a comparison of our derivation with other accounts in Section 5, we close with observations on how this work informs our understanding of modularity and provides future avenues of research in Section 6.

```
;;program
(define-struct pgm [decls body]) ;PGM (id × decl)* × exp

;; declarations
(define-struct procD [ids body])  ;PROC id* × exp
(define-struct globD [])          ;GLOBAL

;; expressions
(define-struct litX [val])            ;LIT val
(define-struct varX [id])             ;VAR id
(define-struct ifX  [test then else]) ;IF exp exp exp
(define-struct seqX [exps])           ;SEQ exp*
(define-struct letX [ids rands body]) ;LET (id × exp)* exp
(define-struct getX [id])             ;GET id
(define-struct setX [id rand])        ;SET id exp
(define-struct appX [id rands])       ;CALL id exp*

(define-struct pcdX [rands])          ;PROCEED exp*
```

Figure 1: Proc Abstract Syntax

2. A PROCEDURAL LANGUAGE – DIRECT SEMANTICS

As with other semantic presentations (e.g. [Wand et al., 2004]), we choose to work with a first-order, mutually recursive procedural language, PROC. Throughout this paper, our systems are given as definitional interpreters, as introduced by Reynolds [1972], in the style of Friedman et al. [2001]. This interpreter-based approach to modeling various AOP mechanisms originated with our work in the Aspect Sandbox[Dutchyn et al., 2002] and related papers[Masuhara et al., 2003, Wand et al., 2004], and was later adopted by others, including Filman [2001]. For this specific paper, this style of presentation emphasizes the reification of continuations as data structures, thus clarifying our specialization claim.

We begin with the usual syntax and direct-style, big-step semantics, given in Figure 1 and Figure 2 respectively. Programs comprise a set of named mutually-recursive, first-order procedures, and a closed, top-level expression. We assume programs and terms are well-typed. Environments are standard.

One important feature of this definition is that we do not specify the order of evaluation for procedure operands. In particular, we use the Scheme `map` procedure to explicitly provide this non-deterministic behaviour.

We should point out that several usual constructs are present in our syntax, but lacking from our evaluator. This does not impair its expressiveness. In particular, the usual constructs are

- (SEQ x_1...) which evaluates each sub-expression in left-to-right order, yielding the value of the last expression, and
- (LET ([i_1 x_1]...) x) which evaluates the body x in an environment enriched with variables i_n bound to the values of the corresponding expressions x_n.

As usual in the literature, these can be denoted in our language the addition of helper procedures as seen in Figure 3. For the sequel, we will employ these notational shorthands.

```
;;; evaluator – expression side
(define (eval x r) ;:(exp × env) → val
  (cond [(litX? x) (litX-val x)]
        [(varX? x) (lookup-env r (varX-id x))]
        [(ifX? x) (eval ((if (eval (ifX-test x) r)
                             ifX-then
                             ifX-else) x) r)]
        [(getX? x) (get-glob (lookup-glob (getX-id x)))]
        [(setX? x) (set-glob (lookup-glob (setX-id x))
                             (eval (setX-rand x) r))]
        [(appX? x) (let ([args (map (lambda (x) (eval x r))
                                    (appX-rands x))]
                         [proc (lookup-proc (appX-id x))])
                     (eval (procV-body v)
                           (extend-env (procV-ids v)
                                       (execF-args f)
                                       empty-env)))]
        [else (error 'eval "not an exp: ~a" x)]))

(define (evlis x* r) ;:(exp* × env) → val*
  (if (null? x*)
      ()
      (cons (eval (car x*) r)
            (evlis (cdr x*) r))))

(define *procs* `([+ . ,(lambda (vs) (+ (car vs) (cadr vs)))]
                  [display . ,(lambda (vs) (display (car vs)) 0)]
                  [newline . ,(lambda (vs) (newline) 0)]))
```

Figure 2: Proc Big-step (Direct) Semantics

(SEQ x_1) ≡ x_1
(SEQ x_1 x_2 ...) ≡ (APP foo i ... x_1)
with helper procedure
(foo . (procV (i ... _) (SEQ x_2 ...)))
where foo is fresh, and each i ... are the free variables of the subsequent expressions x_2 ...
(LET () x) ≡ x
(LET ([i_1 x_1] ... [i_n x_n]) x) ≡ (APP foo i ... x_1 ... x_n)
with helper procedure
(foo . (procV (i ... i_1 ... i_n) x))
where foo is fresh, and each i ... are the free variables of the body x
excluding i_1 ... i_n.

Figure 3: Proc Auxiliary Expressions

```
;;; frames

;; auxiliary
(define-struct testF [then else env]) ;TEST exp exp env :: !bool
(define-struct bindF [ids body env])  ;BIND id* exp env :: !val*
(define-struct randF [exp env])       ;RAND exp env :: !val*
(define-struct konsF [vals])          ;KONS val* :: !val
(define-struct rhsF [id])             ;RHS id :: !val

;; effective
(define-struct getF [])               ;GET :: !loc
(define-struct setF [val])            ;SET val :: !loc
(define-struct callF [id])            ;CALL id :: !val
(define-struct execF [args])          ;EXEC val* :: !proc
```

Figure 4: Proc Small-step (cps) Semantics — Continuations

3. A PROCEDURAL LANGUAGE – CONTINUATION SEMANTICS

In order to identify dynamic join points in a principled way, we need to move to a continuation-passing style (CPS) implementation. Continuations, also known as *goto's with arguments*, were first identified by Strachey [2000] and Landin [1998] to model control flow in programs. Later, Reynolds [1993] applied them to ensure that semantics given by definitional interpreters yields a formal model independent of the defining language control constructs.

The CPS transformation[Danvy and Hatcliff, 1993] of our interpreter is systematic, following closely that of Hatcliff and Danvy [1994]. In essence, we treat each of the `let` expressions in the direct `eval` semantics as a *monadic let*[Moggi, 1989, 1991]. These `lets` express a bind operation between the computation of an operand and the computation awaiting that value. Continuations explicitly sequence these bind operations, and reify the computation awaiting the value.

Usually continuations are presented as closures[Danvy, 2000], but Ager et al. [2005] provide an systematic defunctionalization of these closures into tagged structures and an `apply` procedure that gathers the operations of each closure. The only values that pass into the `apply` operation are object-language values (integers, booleans), lists of object-language values (as argument lists), and references (addresses into the store or to procedures). Each tagged structure must contain the values for each variable that the closures reference. The continuation structures required for our small-step interpreter are given in Figure 4.

As usual in operational semantics, we introduce two *auxiliary* continuations, `randF` and `konsF`, to support multiple arguments to procedures. These two continuations provide a strict right-to-left evaluation order for procedure operands. This choice is arbitrary, as explicitly declared in the direct semantics.[2] We could have supplied a non-deterministic ordering in the CPS semantics, introducing other auxiliary continuations; but, that would distract us from our focus. The essential notion is that these supporting continuations have no basis in the direct semantics: they serve only to bridge the gap between the big-step and small-step systems. A third auxiliary continuation, `rhsF` serves the same purpose with regard to the argument to `setX`.

[2]Recall that `map` in Scheme processes the elements in the list in an explicitly undefined order.

Some formalisms avoid this work by silently introducing products or tuple values. Then a polyadic procedure actually accepts a single tuple argument, and explodes the tuple before evaluation of the body. Similarly, procedure applications would contain a hidden tupling action; paralleling our `konsF` continuation behaviour.

Formal, lambda-calculus approaches eliminate the auxiliary continuations by currying procedures and replace polyadic applications with multiple applications. This simplifies the underlying formalism, allowing development of the soundness proofs of the CPS transformation; Thielecke [1997] provides the details.

For our restricted procedural language, the full power of the λ-calculus is not required. Indeed, in the λ-calculus, the `testF` continuation is unnecessary as well. A simple syntactic transformation makes the consequent clauses into thunks (parameterless closures[Danvy and Hatcliff, 1992]). True and False become binary procedures that simply apply one or the other thunk. In summary, we characterize `randF`, `konsF`, and `testF` as *auxiliary* continuations.[3]

The defunctionalized CPS definition of our interpreter is given in Figures 5 and 6.

Our construction is standard, except in three respects. First, we extend Ager's construction to explicitly linearize the continuation. In Ager's construction, each continuation structure, representing a suspended operation awaiting the value of some expression, would contain the rest of the continuation as a field. Only a halt continuation would not have this, as it has nowhere to continue to. In our construction, we represent the entire continuation as a list of frames. A *frame* is a single element in the list representation of the continuation; it indicates the immediate action when this continuation is activated. The remainder of the continuation is in the tail of the list.

- `push` *:: (frm × cont) → cont* — extends an existing continuation with another frame.
- `pop` *:: (!val × ((frm × cont) → !val)) → cont → !val* — takes a continuation, and either
  - applies the first procedure (`halt`) because the continuation is empty, or
  - applies the second procedure (`step`) to the top continuation frame and the rest of the continuation.

We provide a base definition for `step`, called `base-step` for the language absent aspects. Later we will replace `step` with an aspect-aware version which dispatches appropriately. Also, the halt continuation is represented by the empty list.

The second nonstandard construction is that our implementation *lifts* primitives from the direct interpreter to take the existing continuation as an additional argument. This allows us to provide flow control operations, such as Felleisen's `abort`[Felleisen, 1988], as primitives. This is seen in Figure 7.

Third, our implementation distinguishes the lookup of procedures into a separate continuation, `execF`. Ordinarily, we would require only one continuation, `callF`, to await the evaluation of the operands into argument values. That single continuation would be responsible for locating the de-

[3]These should not be confused with *serious* and *trivial* continuations[Reynolds, 1972], nor with *administrative* continuations[Flanagan et al., 1993].

```
;;; evaluator – expression side
(define (eval x r k) ;: (exp × env × cont) → unit
  (cond [(litX? x) (apply k (litX-val x))]
        [(varX? x) (apply k (lookup-env r (varX-id x)))]
        [(ifX? x) (eval (ifX-test x)
                        r
                        (push (make-testF (ifX-then x)
                                          (ifX-else x)
                                          r)
                              k))]
        [(getX? x) (apply (push (make-getF)
                                k)
                          (lookup-glob (getX-id x)))]
        [(setX? x) (eval (setX-rand x)
                         r
                         (push (make-rhsF (setX-id x))
                               k))]
        [(appX? x) (evlis (appX-rands x)
                          r
                          (push (make-callF (appX-id x))
                                k))]
        [else (error 'eval "not an exp: ~a" x)]))

(define (evlis x* r k) ;: (exp* × env × cont) → unit
  (if (null? x*)
      (apply k
             '())
      (evlis (cdr x*)
             r
             (push (make-randF (car x*) r)
                   k))))

(define (halt v) ;: !val (== val → unit)
  (display v)
  (newline))

(define (apply k v) ;: !(cont × val)
  (((pop halt
         step)
    k)
   v))
```

Figure 5: Proc Small-step (cps) Semantics — Evaluator

```
;;; evaluator – continuation side
(define ((base-step f k) v) ;:(frm × cont) → !val
  (cond ;; auxiliary frames
    [(testF? f) (eval ((if v testF-then testF-else) f)
                      (testF-env f)
                      k)]
    [(randF? f) (eval (randF-exp f)
                      (randF-env f)
                      (push (make-konsF v)
                            k))]
    [(konsF? f) (apply k
                       (cons v (konsF-vals f)))]
    [(rhsF? f) (apply (push (make-setF v)
                            k)
                      (lookup-glob (rhsF-id f)))]

    ;; non-auxiliary frames
    [(getF? f) (apply k
                      (get-glob v))]
    [(setF? f) (apply k
                      (set-glob v (setF-val f)))]
    [(callF? f) (apply (push (make-execF v)
                             k)
                       (lookup-proc (callF-id f)))]
    [(execF? f) (cond [(procV? v)
                       (eval (procV-body v)
                             (extend-env (procV-ids v)
                                         (execF-args f)
                                         empty-env)
                             k)]
                      [(procedure? v) (v (execF-args f) k)]
                      [else
                       (error 'exec "not a procedure: ~a" v)])]
    [else (error 'step "not a frame: ~a" f)]))

(define step base-step)
```

Figure 6: Proc Small-step (cps) Semantics — Evaluator

```
;;; cont ::= frm*

(define (push f k) ;:(frm × cont) → cont
  (cons f k))

(define ((pop e s) k) ;:(!val × ((frm × cont) → !val)) → cont → !val
  (if (null? k)
      e
      (s (car k) (cdr k))))

;;; primitives
(define ((lift p) vs k)
  (apply (p vs) k))

;;; lifted primitives
(define *procs*
  `([+ . ,(lift (lambda (vs) (+ (car vs) (cadr vs))))]
    [cons . , (lift (lambda vs vs))]
    [null? . , (lift (lambda (vs) (null? (car vs))))]
    [display . ,(lift (lambda (vs) (display (car vs)) 0))]
    [newline . ,(lift (lambda (vs) (newline) 0))]
    [abort . ,(lambda (vs k) (apply (car vs) '()))]))

(define (run s)
  (let ([g (parse-prog s)])
    (set! *procs* (cons (PGM-decls g) *procs*))
    (eval (PGM-body g) empty-env '())))
```

Figure 7: Proc Small-step (cps) Semantics — Primitives

sired procedure and initiating the evaluation of it's body-expression with the desired bindings.

Examining the direct semantics closely, we can see that there are two `let` bindings present in the case of an `APP` expression. Other one-step[Danvy and Nielson, 2003] and A-normal[Flanagan et al., 1993] transformations optimize portions of this transformation, usually the second binding. Our more naïve approach allows us to expose the two separate operations, which will be valuable as we extend the system to incorporate dynamic join points, pointcuts, and advice.

4. EXPOSING AOP CONSTRUCTS

With these preliminaries, we are prepared to expose the latent dynamic join points in PROC, and provide syntax to denote pointcuts and advice. We need to describe three items (quoted from [Kiczales et al., 1997]):

1. **dynamic join points** — "principled points in the execution". These will be states in the interpreter where values are applied to non-auxiliary continuation frames.

2. **pointcuts** — "a means of identifying join points". These will be syntax for predicates over the value and continuation frame content.

3. **advice** — "a means of affecting the semantics at those join points". This is implemented as the advice body as a procedure applied to the continuation frame.

We will examine each of these in turn.

4.1 Dynamic Join Points

Dynamic join points are the first abstraction in our model. Other semantic models simply list dynamic join points without supporting the intuition for their selection. The underlying principles are not enunciated. Identifying this principle is a key result of this work.

For us, join points are activations of certain continuation frames. Recall that we introduced auxiliary frames to support our eager, right-to-left evaluation order in the CPS semantics. Therefore, we adopt the following principle:

> A dynamic join point is modeled as a state in the interpreter where a non-auxiliary continuation frame is applied to a value.

Auxiliary continuation frames do not correspond to principled points in the execution of a program. For example, our `konsF` and `randF` frames were arbitrarily chosen to supply an eager, right-to-left evaluation order. With a lazy big-step semantics, or with a different evaluation order, different auxiliary continuation frames would be required. Similarly, the `testF` frame exists to postpone the choice of alternatives to an `ifX` until the test has been evaluated first. The `rhsF` and bindF auxiliary frames exist to support the single reduction ordering that CPS interpreters must support – again they are not mandated by the big-step semantics.

Therefore, in PROC, we have four frames corresponding to dynamic join points:

- `callF` ($id \vdash !val^*$)) — takes an procedure name and constructs a frame that will consume a list of argument values and apply the named procedure to them,

- `execF` ($val^* \vdash !proc$) — stores a list of argument values and constructs a frame that will consume a procedure and apply it to the list of values,

- `getF` ($id \vdash !loc$) — takes an identifier and constructs a frame that will consume a store location and continue with its content,

- `setF` ($val \vdash !loc$) — takes a value and constructs a frame that will consume a store location and continue after updating its content.

The type signatures indicate the type of the information stored in the continuation frame, followed by type of the continuation once the frame is `push`ed. We use negative types for continuations, in keeping with previous workJouvelot and Gifford [1989], Murthy [1992]. Thielecke [1997] explores this in detail.

In each case, a dynamic join point has various items of information available, some from the value applied to the continuation, some from the frame itself. These include

1. a procedure, either by name (in the case of `callF`) or as an actual structure (in the case of `execF`),

2. a list of values corresponding to the arguments to the procedure (in the case of `callF` or `execF`,

3. a value and a store reference (in the case of `setF`),

4. a store reference (in the case of `getF`).

Our join points are summarized in Table 1.

In our model, dynamic join points make accessible the latent control structure of the language semantics. Dynamic

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

Table 1: Dynamic Join Points

```
;;; pointcuts

;; effective continuation frame matching
(define-struct getC [gid])        ; GETPC id
(define-struct setC [gid id])     ; SETPC id id
(define-struct callC [pid ids])   ; CALLPC id id*
(define-struct execC [pid ids])   ; EXECPC id id*

;; combinational
(define-struct orC [pcs])         ; ORPC pcut*
(define-struct notC [pc])         ; NOTPC pcut
```

Figure 8: Proc Pointcuts — Abstract Syntax

join points correspond to continuation frames, and are modeled by states within the interpreter. Our set of dynamic join points is stable with regard to semantic changes such as altering the order of evaluation, or moving from eager to lazy evaluation.[4] Other semantic changes involved in extending the big-step semantics, notably introducing new terms (e.g. for-loops[Harbulot and Gurd, 2006]), would introduce or modify the set of dynamic join points.

Our dynamic join points systematically align with points in the model that are well-accepted as being semantically meaningful. Our principle defines this systematic alignment. In other models, some have framed dynamic join points as program rewrite points[Aßmann and Ludwig, 1999, Roychoudhury and Gray, 2005]. Other accounts have dynamic join points appear as an ad hoc list, including in our earlier work[Wand et al., 2004]. Our principled approach provides a more robust and elegant description.

4.2 Pointcuts

The second abstraction we must add to our model is pointcuts. Pointcuts are syntax that provide a means to identify our dynamic join points. We have a pointcut for identifying each kind of continuation frame (join point): `call`, `exec`, `get`, and `set`. We adopt the following syntax for pointcuts. It contains four structures, one for each kind of dynamic join point.

We have chosen a direct pointcut syntax, where the procedure name and the argument names are given directly in the pointcut. In the next section, we will use the argument names to offer access to the arguments in the advice. The semantics of a pointcut is to examine whether the current interpreter state matches the identified continuation frame – both in kind and content – and the current value. This is seen in Figure 9.

In the case of a `callC` pointcut, we ensure that the frame is a `callF` frame, and that it holds a procedure name equal

[4]Changing to lazy evaluation would alter the order that join points are encountered during the evaluation.

```
;;; matching
;MATCH id* val* (val* → (val × frm))
(define-struct match [ids vals prcd])

(define (match-pc c v f) ;:(pcut × val × frm) → match
  (cond ;; combinational pointcuts
        [(orC? c) (let loop ([pcs (orC-pcs c)])
                    (if (null? pcs)
                        #f
                        (or (match-pc (car pcs) v f)
                            (loop (cdr pcs)))))]
        [(notC? c) (if (match-pc (notC-pc c) v f)
                       #f
                       (make-match '()
                                   '()
                                   (lambda (nv)
                                     (values v f))))]
        ;; fundamental pointcuts
        [(getC? c) (and (getF? f)
                        (eq? (lookup-glob (getC-gid c)) v)
                        (make-match '()
                                    '()
                                    (lambda (nv)
                                      (values v f))))]
        [(setC? c) (and (setF? f)
                        (eq? (lookup-glob (setC-gid c)) v)
                        (make-match `(,(setC-id c))
                                    `(,(setF-val f))
                                    (lambda (nv)
                                      (values v
                                              (make-setF
                                               (car nv))))))]
        [(callC? c) (and (callF? f)
                         (eq? (callC-pid c) (callF-id f))
                         (make-match (callC-ids c)
                                     v
                                     (lambda (nv)
                                       (values nv f))))]
        [(execC? c) (and (execF? f)
                         (eq? (lookup-proc (execC-pid c)) v)
                         (make-match (execC-ids c)
                                     (execF-args f)
                                     (lambda (nv)
                                       (values v
                                               (make-execF
                                                nv))))))]
        [(advC? c)]
        [else (error 'match-pc "not a pointcut: ~a" c)]))
```

Figure 9: Proc Pointcuts — Implementation

to the one given in the pointcut. For a `execC` pointcut, we ensure that the frame is an `execF` frame and that the supplied value is a procedure whose name is equal to the one given in the pointcut. `GetC`, and `setC` pointcuts are similar, matching the `getF` and `setF` frames respectively.

We also include two combinational pointcuts. The first is `orC`, which matches any dynamic join point which matches the first subpointcut; or, failing that, matches the second subpointcut. This allows us to abstract a concern that cuts across multiple procedures. For example, one might consider two `displayX` procedures, each with a different output format, to be a single display concern.

This combinational pointcut provides a simple specialization ordering to pointcuts; and, by extension, advice. Any given pointcut, `A`, is more specialized than `orC(A B)` for any distinct `B` pointcut. Pointcuts do not have a unique total ordering, only a partial order. They can be totally ordered using the standard topological sort. By extension, advice can be ordered by this total pointcut order.

The other combinational pointcut is `notC` which simply matches every join point which differs from its subpointcut. It returns no matched values, it simple succeeds or fails.

A pointcut matches the top continuation frame, the list of identifiers from the pointcut is returned. If a match is not found, `#f` (Scheme `false`) is returned. In our implementation, matching against a `orC` pointcut yields the identifiers for the matching sub-pointcut. This means that each subpointcut must provide the same identifiers.

In our model, we adopt the principle that

> pointcuts do not alter the semantic behaviour of the program or language.

In our system, advice is solely responsible for altering behaviour at join points. This leads to concerns with contextual pointcuts.

4.2.1 Contextual Pointcuts

It is tantalizing to consider the entire continuation for the purposes of matching join points. If we did this, then we can quickly and easily provide the various contextual pointcuts, including a novel one:

- (`cflowbelow pcut`): climb down (towards older) the list of frames, skipping the current frame,
- (`cflow pcut`): equivalent to (`or pcut` (`cflowbelow pcut`)).
- (`cflowabove pcut`): climb back up (towards newer) the list of frames, skipping the current frame.[5,6]

These contextual pointcuts provide a mechanism for characterizing join points based on their temporal context in the control flow. The usual `cflow` and `cflowbelow` provide the usual "within another control context" recognizer by searching downward toward the program start. Our novel `cflowabove` construct provides a way to search in the other direction, "encloses another control context". This is useful, along with the `not` pointcut, to provide the equivalent to Prolog cuts in the context search.

Unfortunately, in a language with tail call optimization, this simplistic implementation does not work. The context of interest may be removed from the continuation frame list by the tail call optimization, and the desired advice will not be triggered. In fact, deeper consideration of the contextual pointcuts convinces us that these pointcuts actually have a computational effect: they require the evaluator to remember where the exit from the interesting context occurs. This is conveniently simple in non-tail call languages: popping the identified continuation frame can serve as the marker.

If we know the pointcuts in advance, we can avoid having the pointcut alter all matching frame behaviour by retaining only the interesting frames, the ones identified in `cflow` pointcuts, on the stack. This requires advance knowledge; but can then be implemented quite efficiently[Clements and Felleisen, 2004].

But, tail call languages require some additional mechanism — context may disappear before related advice is triggered. One solution might be to include some special context-marking continuation frame – these are called *continuation marks*[Clements and Felleisen, 2004], and essentially provide a safe-for-space implementation of dynamic binding. This is the mechanism applied in the AspectScheme language[Dutchyn et al., 2006].

We choose not to add new mechanisms, and wish to cleave to the principle that pointcuts do not change the language semantics nor the program behaviour. Therefore, we must supply separate implementations of these pointcuts. Fortunately, Masuhara et al. [2003] provides a state-based `cflow` design. It can be modelled in our language as two coordinated pieces of advice. The first specialises join points matching the control flow of interest to push the arguments onto a stack data structure, `proceed` to determine the result, pop the stack, and return that result. The second specialises the join points of interest within the control flow to check for available context on the stack data structure, and modify the continuation behaviour appropriately.

In our model, pointcuts are first-order predicates for dynamic join points. In this general view, we are no different from other accounts of dynamic join points, pointcuts, and advice AOP. But, pointcuts identify continuation frames at which advice bodies are to operate. Hence, we can view advice as extending and specializing the behaviour of control points in programs.

4.3 Advice

Now we come to the third feature of our model — advice. A piece of advice needs to specify a means of affecting the semantics at join points. Syntactically, it contains two parts:

1. a pointcut — which indicates which dynamic join points are to be affected
2. an advice body — an expression

The new syntax element for advice declarations is given in Figure 10. Advice are declarations in our model, just like procedures. Therefore, they will be have identifiers bound to them, just like procedures do.

[5]With cactus stacks[Clinger et al., 1999] for threaded languages, this requires the correct path back up to be maintained.

[6]Pointcuts containing `cflowabove` can be rewritten using `cflowbelow` alone. This is clear by recognizing that pointcuts form a regular language describing stack structures[Sereni and de Moor, 2003], and that `cflowbelow` and `cflowabove` are the left- and right-regular descriptions. Of course, `cflowabove` is expressive in the sense of Felleisen [1991] because the tranformation requires a global rewrite of the entire pointcut.

```
;;; declarations
(define-struct advD [pc body]) ;DECL +:= ADVICE pcut exp
```

Figure 10: Proc Advice Declaration – Abstract Syntax

```
(BEFORE pcx) ≡ (AROUND pc (SEQ x proceed))
(AFTER pc x) ≡ (AROUND pc (APP foo (proceed)))
with fresh helper procedure
(foo . (procV (v) (SEQ x v)))
```

Figure 11: Proc Before and After Advice

In our system, all advice is `around` advice. That is, it has control over, and alters the behaviour of, the underlying dynamic join point. Our advice may `proceed` that dynamic join point zero, one, or many times. This does not restrict the generality of our model, as common `before` and `after` advice are the two possible orderings of the advice body and `proceed`, shown in Figure 11.

Semantically, an advice resembles a procedure. The pointcut part identifies the affected dynamic join points, and provides binding names for the arguments of the dynamic join point. In our model the advice body acts like a procedure body, but its locus of application differs.

A procedure is usually applied to some values to yield another value. For example, the procedure `pick` in the following code:

```
(define (pick b) (if x 1 2))
(+ (pick #t) 3)
```

is applied to `#t` to yield a new value `1`. Filinski [1989] first recognized that `pick` transforms the continuation of the procedure application from

```
(lambda (n)   ; await number, add three, halt
  (+ n 3))
```

to

```
(lambda (b)   ; await boolean
  (let ([n (if b 1 2)])   ; select number
    ((lambda (n) (+ n 3)) ; original continuation
     n)))      ; given the selected number
```

One way to discern this different mode of application is to consider the types of the elements involved. Jouvelot and Gifford [1989] recognized that the type of the original continuation is !number (read as consumes number), and that applying `pick` has extended the continuation to consume a boolean (typed !boolean). `Pick` has type boolean → number when considered as a value transformer, and has type !number →!boolean as a continuation transformer[Strachey, 2000].

In Filinski's *symmetric lambda calculus*[Filinski, 1989], procedures could be applied in either way: to values, yielding new values; or to continuations, yielding new continuations. In our model, advice provides this similar procedure application to continuations. We present our semantics in five parts – advice elaboration and matching, altered `step/prim` to support advice execution, a new `step/weave` to weave advice into the execution of the program, advice invocation, and last, the `proceed` expression.

First, we recognize that advice is a declaration; hence we need to elaborate the advice declarations, in the same trivial way we did for procedure declarations. This is displayed in Appendix A.

Matching is also shown in Figure 12. We simply walk the elaborated list of advice, comparing the pointcuts and returning a `match` containing the pointcut-match identifiers and the advice itself. It also contains details on how to `proceed`, but we will examine those later.

Pointcuts not only provide parameters at the application site, but also automate the application of advice to all matching dynamic join points. This universal application of advice extends the semantics of matching dynamic join points to contain additional behaviour.

A subtle difference is that advice can extend the behaviour of a join point, by calling `proceed`, a new expression in our PROC language. It takes a set of arguments and passes them on to the next advice, or the underlying dynamic join point if all advice has been invoked. The syntax for `proceed`, as well as the extension of `eval` is given in Figure 13.

In order for `proceed` to work, we need to provide the remaining matched advice, and a representation of the original join point. This is done by binding a special variable, '`%proceed` into the environment for the advice. It contains the remaining advice, if any, and the original procedure name (in the case of a `callF` dynamic join point), the original `procV` or procedure (in the case of an `execF` dynamic join point).

Recalling our principle that dynamic join points correspond to frame activations, we recognize that our new frame, `advF` defines a new set of dynamic join points that may be matched against. By construction, all of our declarations are bound to identifiers, advice declarations will also have names. Hence, we naturally provide an advice execution dynamic join point, and its associated matching operation. By construction, all activations of `advF` frames are processed by `adv-step`, so the weaving of additional behaviour is automatic. The call structure that makes this so is:

- `apply` calls `adv-step`
- `adv-step` looks for matching advice
  - if there is none, `base-step` provides the fundamental behaviour of the dynamic join point
  - if there are matches, we evaluate arguments and push an advice execution dynamic join point

```
;;advice matching against frames/join points
(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)
                   k))]
          [else (loop (cdr advs))])))

(define step adv-step) ;;redefinition
```

Figure 12: Proc Advice – Matching

```
;;; proceed needs a new advice-execution frame
;; – hence, a new join point
ADV (val* rightarrow val+frm) adv* :: !val
(define-struct advF [v->v+f advs])

;;; evaluator – expression side
(define (eval x r k) ;:(exp × env × cont) → unit
  (cond ...
        [(pcdX? x) (evlis (pcdX-rands x)
                          r
                          (push
                           (make-advF (lookup-env r '%proceed)
                                      (lookup-env r '%advs))
                           k))]
        [else (error 'eval "not an exp: ~a" x)]))

;;; evaluator – continuation side
(define ((base-step f k) v) ;:(frm × cont) → !val
  (cond ...
        ;; non-auxiliary frames
        ...
        [(advF? f)
         (let-values ([(v1 f1) ((advF-v->v+f f) v)])
                     (((adv-step (advF-advs f)) f1 k) v1))]
        [else (error 'step "not a frame: ~a" f)]))
```

Figure 13: Proc Advice – Proceed

- `proceed` expressions do the same to extract the next advice or the final dynamic join point and initiate it's execution.

In our model, an advice body provides new behaviour for each dynamic join point (control point) identified by the advice' pointcut. This new behaviour *extends* the original because advice may contain additional program operations. This new behaviour *specializes* the original because the original behaviour is available through the `proceed` expression.

5. COMPARISON TO OTHER SEMANTICS

We compare our dynamic join point schema to those of other semantic models. The first two are semantic models are joint work between this author and others.

5.1 Aspect Sandbox

In joint work, Dutchyn et al. [2002] and Wand et al. [2004], this author developed a number of semantic models of aspect-oriented programs, both for object-oriented and procedural languages. That work provides a model of a first-order, mutually-recursive procedural programming language. In that semantic model, three kinds of dynamic join points were constructed ex nihilo: `pcall`, `pexecution`, and `aexecution`. This work develops the principle behind the intuition of those three dynamic join point kinds.

Our model also eliminates some of the irregularities in these other implementations. For instance, because Wand et al. [2004] implements a direct semantics, it maintains a separate stack of dynamic join points rather than relying on structured continuations to do this. Further, it relies on thunks to delay execution of `proceed`; in our semantics, this arises within from the continuation structure.

We focus on the core semantic model for our system, therefore we have avoided the more extensive pointcut languages found in mainstream languages. We adopt conventions from early versions of AspectJ[Kiczales et al., 2001]. Current version of AspectJ provides a pointcut calculus with separate binding combinators (e.g. `args`, and `target`), as well as pattern matching and other features. In our model, `&&` provides no additional expressive power, so we do not include it.

Some Aspect Sandbox pointcuts are lexical, such as `within` which restricts join point matches to those which occur during the evaluation of the expressions within a specific procedure. It can be characterized as join points which appear with no intervening frames; as this is the situation where lexical and dynamic scoping coincide. But, this pointcut is strongly dependent on the textual representation of the program. As a result, programmers can easily re-modularize or abstract their code to retain what appears to be the same join point sequencing; but unwittingly introduce or eliminate join points from those identified by this pointcut.

`Within` is dramatically at odds with the dynamic join point and advice model. Indeed, it can mislead programmers into believing that dynamic join points are expressions and that aspect-oriented programming is simply a program generation/rewriting technique. Our model shows how join points and advice aspects arise from the *semantics* of a language, not from the *syntax* of a language. `Within` is possible wiht our framework, based on the observation that lexical scope coincides with dynamic scope, until another lexical scope intervenes. So, `within` can be implemented similar to our `cflow` example, with a third piece of advice that masks the stack data structure once another lexical scope is entered. The join points identifying a new scope are the execution join points; available through the `exec` pointcut.

In summary, our pointcut language provides is a reasonable fit for our model approach.

5.2 AspectScheme

The author contributed the semantic description of AspectScheme[Dutchyn et al., 2006] and the online implementation[Dutchyn, 2006]. AspectScheme models join points as procedure applications in context of other in-progress procedure applications. It depends on novel *continuation marks* to express the structure of the continuation stack, and relies on macros to provide weaving whenever a procedure is applied. This is practical solution for extending Scheme, where continuations are available only as opaque procedures—their structure cannot be examined. This work simplifies the AspectScheme semantic presentation to recognize that continuation marks are not required, provided Ager et al. [2005]'s defunctionalized continuation model is available.

AspectScheme offers only a single kind of dynamic join point, a procedure application in the context of pending procedures. This corresponds to our `execF` dynamic join point, but with additional context. But, because dynamic join points are first-class objects, temporally ordered lists of procedures and arguments in AspectScheme, the programmer can extend the set of pointcuts by writing their own. This expressiveness is put to good use in showing practical applications of advice.

5.3 PolyAML and μABC

Dantas et al. [2005] provide a PolyAML, a polymorphic aspect-oriented programming language. It is implemented in two levels, a polymorphic surface syntax, which is translated into a monomorphic dynamic semantics, $\mathbb{F}_A$. Their focus is on type-checking, and `around` aspects are incompatible with that goal. They can only support oblivious[Filman and Friedman, 2004] aspects, which must be `before` and `after` only. A later paper,[Dantas et al., to appear] solves the typing difficulties with `around` advice, using novel local type inference techniques.

Their monomorphic machine is described in terms of context semantics[Ager et al., 2003]. Briefly, a context is an expression with a hole, which the current redex will plug, once it reduces to a value. The machine shifts into deeper and deeper contexts until values can be directly computed, either as literals or variable references. Once all the holes are plugged in a redex, it is reduced to a value and plugged into its pending context. Danvy et al. investigated the equivalence between context semantics and continuation semantics. PolyAML's label method for providing aspects in monomorphic context semantics appears to be equivalent to AspectScheme's continuation marks in a continuation semantics.

It would be interesting to attempt to remove the labels from their $\mathbb{F}_A$ core calculus by reifying the actual continuation structures. We expect that the principled set of dynamic join points would again become apparent, rather than imposed externally.

Bruns et al. [2004] provides an untyped core calculus for aspects. As Dantas et al. [2005] note, this core calculus strongly resembles their $\mathbb{F}_A$ monomorphic context semantics. Again, labels are used to annotate a context and provide an understanding of dynamic join points. They support full `around` advice, but make no attempt to supply static type checking or inference.

5.4 Other Related Work

Several other semantic formulations for aspects have been offered.

Douence et al. [2001] considers dynamic join points as events, and provides oblivious aspects. This is done by providing a custom sequencing monad that recognizes computations, and wraps them with the additional behaviour of the advice. Unfortunately, this is insufficient to allow `around` advice to alter the parameters of the wrapped computation. Only the option to `proceed` with the original arguments is available.

Andrews [2001] provides a process-calculus description of aspects. Oblivious aspects are provided. But, constrained by encapsulated processes, full `around` aspects are not possible.

Clifton and Leavens [2006] further explored the idea of split `call` and `exec` join points; a distinction that originated in Wand et al. [2002].

Kojarski and Lorentz [2005] consider composition of multiple aspect extensions to a base language. Our work indicates that, for pointcut-and-advice aspects, there seems to be a natural extension, which modularizes over the dynamic semantic of the language. Preliminary work suggests that the same principled approach can be applied to other phases [Cardelli, 1988] in a programming language, leading to several other extensions such as static join points (as in AspectJ). The compositional behaviour of these different phases remains to be explored.

Endoh et al. [2006] also use the CPS transformation to expose join points; their interest is to reduce the number of advice kinds (e.g. `after returning`). This work aims at a more fundamental understanding of pointcuts and advice AOP, and attempts to expose the principles underlying it. As a result, our work identifies an additional join point, the advice-execution join point (which AspectJ provides without explanation). Furthermore, we takes view that advice specialize continuation behaviour, leading to a principled understanding of the modularity that AOP can provide.

6. SUMMARY

Aspect-oriented programming (AOP) is crosscutting-modularity technology. It comes in a variety of forms, including open classes and dynamic join points. The former example provide separation of concerns that involve data modularity. This paper demonstrates that the latter provides separation of concerns invested in modularizing (identifying, specializing and isolating) control structures. Although unsurprising, this characterization of different kinds of aspects based on what they modularize is powerful. It fundamentally sharpens our understanding *what dynamic aspects are*, and therefore enables us to construct and apply them effectively.

Our construction has intriguing parallels with object-oriented programming. From one perspective, objects provide a way for programmers to group related data fields, tagging them so that late-bound operations can be supplied. Our construction appears to generalize to grouping related continuation frames, tagging them so that late-bound operations can be supplied.

One example of this sort of aspect hierarchy would be a base aspect that provides the state-based implementation of `cflow`. By extending that abstract aspect with two pointcuts and the desired advice, we provide modular instances of `cflow` and its ilk; thus eliminating them from the base language. The language remains expressive and becomes simpler.

The duality between values and continuations may offer some insight into what dynamic join point-based AOP is effective at modularizing. We believe that a full-fledged aspect, intended to capture a crosscutting feature, combines multiple pointcuts and advice into an *abstract control type* paralleling ubiquitous abstract data type. Our implementation highlights the existence of a dispatch to late-bound advice specializing the behaviour of that continuation frame. Can this be extended to give a unified understanding of aspects and objects as dual similar to value and continuation duality?

Furthermore, our construction suggests that an appropriate type theory for dynamic join points should be built on the ones for continuations. In particular, we are investigating the negative types of [Griffon, 1990, Murthy, 1992] which characterize continuations, and the more recent work of Shan [2003] and Biernacki et al. [2006] who look at polarized types for delimited continuations[Biernacka et al., 2005, Shan, 1999] of which our frames are a degenerate example.

In summary, our work provides a well-founded implementation of aspects with three key properties:

1. Dynamic join points, pointcuts, and advice aspects are modeled directly in continuation semantics; without the need for extraneous labels or continuation marks,

2. Principled dynamic join points arise naturally, as continuation frames, from describing programming languages in continuation semantics, and

3. Advice acts as a procedure on these continuation frames, providing specialized behaviour for them.

We give a formal model of dynamic join points, pointcuts, and advice built on the well-understood processes of conversion to continuation-passing-style, and defunctionalization. We demonstrate that dynamic join points arise naturally in this formulation, as continuation frames. Therefore, advice can specialize their behaviour directly in our construction. Furthermore, we demonstrate that, in our model, `cflow` corresponds to a continuation context, and interacts poorly with tail-call optimizations, but can be recognized as a state effect.

In this way, we provide a fundamental account of these AOP mechanisms that arises naturally from the semantic description of the language. Our model is by construction, not ad hoc. Our model does not entail pre-processing or other meta-programming techniques.

References

M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *International Conference on Principles and Practice of Declarative Programming*, pages 8–19. ACM Press, Aug. 2003. ISBN 1-58113-705-2.

M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005.

J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In A. Yonezawa and S. Matsuoka, editors, *Workshop on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192, pages 187–209. Springer-Verlag, Sept. 2001. ISBN 3-540-42618-3.

A. Aßmann and A. Ludwig. Aspect weaving by graph rewriting. In U. Eisenecker and K. Czarnecki, editors, *International Symposium on Generative Component-based Software Engineering*, Oct. 1999.

M. Biernacka, D. Biernacki, and O. Danvy. An operational foundation for delimited continuations in the cps hierarchy. Technical Report RS–05–25, BRICS, University of Aarhus, Aug. 2005. URL `http://www.brics.dk/RS/05/24/BRICS-RS-05-24.pdf`. to appear in Logical Methods in Computer Science.

D. Biernacki, O. Danvy, and C. chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60(3):274–297, 2006.

G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. μABC: A minimal aspect calculus. In P. Gardner and N. Yoshida, editors, *International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224. Springer-Verlag, Sept. 2004. ISBN 3-540-22940-X.

L. Cardelli. Phase distinctions in type theory. Manuscript, 1988. URL `citeseer.ist.psu.edu/cardelli88phase.html`.

J. Clements and M. Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems*, 26(6):1029–1052, 2004.

C. Clifton and G. T. Leavens. MiniMAO: An imperative core language for studying aspect-oriented reasoning. *Sci. Comput. Programming*, 63(3):321–374, Dec. 2006.

W. D. Clinger, A. H. Hartheimer, and E. M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, Apr. 1999.

Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. *Structuring Operating System Aspects*, chapter 28, pages 651–657. In , Filman et al. [2004], Oct. 2004.

D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. PolyAML: A polymorphic aspect-oriented functional programming language. In O. Danvy and B. C. Pierce, editors, *International Conference on Functional Programming*, Sept. 2005. ISBN 1-59593-064-7.

D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *Transactions on Programming Languages and Systems*, to appear.

O. Danvy. Formalizing implementation strategies for first-class continuations. In G. Smolka, editor, *European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 88–103. Springer-Verlag, Mar. 2000. ISBN 3-540-67262-1.

O. Danvy and J. Hatcliff. Thunks (continued). In *Workshop on Static Analysis*, pages 3–11, 1992.

O. Danvy and J. Hatcliff. On the transformation between direct and continuation semantics. In S. D. Brookes, M. G. Main, A. Melton, M. W. Mislove, and D. A. Schmidt, editors, *Conference on Mathematical Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 627–648. Springer-Verlag, Apr. 1993. ISBN 3-540-58027-1.

O. Danvy and L. R. Nielson. A first-order one-pass cps transformation. *Theoretical Computer Science*, 308(1–3): 239–257, Nov. 2003.

B. De Win, W. Joosen, and F. Piessens. *Developing Secure Applications Through Aspect-Oriented Programming*, chapter 27, pages 633–560. In , Filman et al. [2004], Oct. 2004.

R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Lecture Notes in Computer Science*, volume 2192, pages 170–186, Sept. 2001.

C. J. Dutchyn. AspectScheme v. 2. PLaneT repository, Jan. 2006. URL `http://planet.plt-scheme.org/300/#aspect-scheme.plt2.1`.

C. J. Dutchyn, G. Kiczales, and H. Masuhara. Aspect Sandbox. internet, 2002. URL `http://www/labs/spl/projects/asb.html`.

C. J. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, Dec. 2006.

Y. Endoh, H. Masuhara, and A. Yonezawa. Continuation join points. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *Workshop on Foundations of Aspect Oriented Languages*, pages 1–10, Mar. 2006. Iowa State University TR#06-01.

M. Felleisen. The theory and practice of first-class prompts. In *Symposium on Principles of Programming Languages*, pages 180–190, 1988.

M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

A. Filinski. Declarative continuations and categorical duality. Master's thesis, DIKU, University of Copenhagen, Aug. 1989.

R. Filman. Understanding AOP through the study of interpreters, 2001. URL `citeseer.ist.psu.edu/571298.html`.

R. Filman and D. Friedman. *Aspect-Oriented Programming is Quantification and Obliviousness*, chapter 2, pages 21–36. In , Filman et al. [2004], Oct. 2004.

R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Oct. 2004.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Conference Programming Language Design and Implementation*, pages 237–247, 1993.

D. Friedman, M. Wand, and C. Haynes. *Essentials of Programming Languages*. MIT Press, 2001.

T. G. Griffon. A formulæ-as-types notion of control. In *Symposium on Principles of Programming Languages*, pages 47–57. ACM Press, January 1990.

B. Harbulot and J. R. Gurd. A join point for loops in AspectJ. In R. Filman, H. Masuhara, and A. Rashid, editors, *Conference on Aspect Oriented Software Development*, pages 63–74. ACM Press, Mar. 2006. ISBN 1-59593-300-x.

J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *Symposium on Principles of Programming Languages*, pages 458–471, 1994.

P. Jouvelot and D. K. Gifford. Reasoning about continuations with control effects. In R. L. Wexelblat, editor, *Conference Programming Language Design and Implementation*, pages 218–226. ACM Press, June 1989. ISBN 0-89791-306-X.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, June 1997.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, 2001.

S. Kojarski and D. H. Lorentz. Pluggable AOP – designing aspect mechanisms for third-party composition. In R. P. Gabriel, editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 247–263. ACM Press, Oct. 2005. ISBN 1-59593-031-0.

P. J. Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2):125–143, 1998.

H. Masuhara, G. Kiczales, and C. J. Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, January 2003.

E. Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science*, pages 14–23. IEEE, June 1989.

E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

C. R. Murthy. A computational analysis of girard's translation and LC. In *Symposium on Logic in Computer Science*, pages 90–101. IEEE, June 1992.

J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM National Conference*, pages 717–740. ACM Press, 1972.

J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993.

S. Roychoudhury and J. Gray. AOP for everyone – cracking the multiple weavers problem. Manuscript, 2005. URL `http://www.cis.uab.edu/gray/Pubs/software-suman.pdf`.

D. Sereni and O. de Moor. Static analysis of aspects. In *ACM SIGPLAN Conference on Aspect-oriented Software Development*, pages 30–39, 2003.

C.-c. Shan. From shift and reset to polarized logic. Manuscript, 2003. URL `http://www.eecs.harvard.edu/čcshan/polar/paper.pdf`.

C.-c. Shan. Shift to control. In O. Shivers and O. Waddell, editors, *Scheme Workshop*, 1999.

O. Spinczyk and D. Lohmann. Using AOP to develop architecture-neutral operating system components. In *SIGOPS European Workshop*, pages 188–192. ACM Press, Sept. 2004.

C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13 (1/2):11–49, 2000.

H. Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, 1997. Also available as technical report ECS-LFCS-97-376.

M. Wand, G. Kiczales, and C. J. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In G. T. Leavens and R. Cytron, editors, *Workshop on Foundations of Aspect Oriented Languages*, pages 1–8, Apr. 2002. Iowa State University TR#2-06.

M. Wand, G. Kiczales, and C. J. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *Transactions on Programming Languages and Systems*, 26(4):890–910, Sept. 2004.

APPENDIX

A. PROC ELABORATOR

```
;;; Elaborator

(define *globs* #f) ;; (id × boxed-val)*
(define *procs* #f) ;; (id × proc/prim)*
(define *advs* #f)  ;; (pc × adv)

;; values – val ::= constant — procedure
(define-struct procV [ids body]) ;; PROC id* exp

(define _init-val_ 0) ;; val

;; location LOC ::= ref val (ie. box)

(define (lookup-glob i) ;; id → loc
  (let ([i+b (assq i *globs*)])
    (if i+b
        (cadr i+b)
        (error 'glob "not found: ~a" i))))

(define (lookup-proc i) ;; id → proc
  (let ([i+p (assq i *procs*)])
    (if i+p
        (cadr i+p)
        (error 'proc "not found: ~a" i))))

(define (get-glob l) ;; loc → val
  (unbox l))

(define (set-glob l v) ;; (loc × val) → val
  (let ([ov (unbox l)])
    (set-box! l v)
    ov))

(define ((lift o) v* k) ;; (val* → val) → (val* × cont) → !val
  (apply k (o v*)))

(define (elab! prims i+d*) ;; ((id × !(val* × cont))* × (id × decl))* → unit
  (set! *globs* '())
  (set! *procs* prims)
  (set! *advs* '())
  (for-each (lambda (i+d)
              (let ([d (cdr i+d)]
                    [i (car i+d)])
                (cond [(procD? d) (set! *procs* `((,i ,(make-procV (procD-ids d)
                                                                   (procD-body d)))
                                                  . ,*procs*))]
                      [(globD? d) (set! *globs* `((,i ,(box _init-val_))
                                                  . ,*globs*))]
                      [(advD? d) (set! *advs* `((,(advD-pc d) . ,(advD-body d))
                                                . ,*advs*))]
                      [else (error 'elab "not a decl: ~a" d)])))
            i+d*)
  (set! step (adv-step *advs*)))
```

Figure 14: Proc Elaborator