Dynamic Join Points:

Model and Interactions

by

CHRISTOPHER J. DUTCHYN

Bachelor of Science, (*Mathematics* (*Honours*)), University of Alberta, 1989
Master of Science, (*Computing Science*), University of Alberta, 2002

A DISSERTATION SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

November 2006

© Christopher J. Dutchyn, 2006

Abstract

By modeling dynamic join points, pointcuts, and advice in a continuation-passing style interpreter, we provide a fundamental account of these AOP mechanisms. This account frames interesting type-and-effect properties of the mechanisms, such as the range of interactions between advised code and advice, and provides a general framework for describing these aspect interactions.

Contents

Abstract . . . . *ii*

Contents . . . . *iv*

Tables . . . . *viii*

Figures . . . . *x*

Preface . . . . *xii*

Acknowledgments . . . . *xiv*

Dedication . . . . *xvi*

1 Introduction **1**
- 1.1 Motivation 2
  - Thesis 2
  - Contributions 2
- 1.2 Plan of Presentation 4

I Dynamic Semantics **5**

2 A Model for Dynamic Join Points, Pointcuts, and Advice . . **7**
- 2.1 A Procedural Language – Direct Semantics 8
- 2.2 A Procedural Language – Continuation Semantics 9
- 2.3 Exposing Our AOP Constructs 14

2.4 Comparison to Other Semantics . . . 24
Aspect Sandbox . . . 24
AspectScheme . . . 25
PolyAML and uABC . . . 26
Other Related Work . . . 27
2.5 Summary . . . 27

3 Advice in Higher Order Languages . . . 29
3.1 AspectScheme Model . . . 29
Background on the CEKS machine . . . 30
Declaring Advice . . . 31
Function Equality . . . 33
Primitive Function Application . . . 35
Regular Function Application . . . 37
3.2 AspectScheme with *Cflow* . . . 38
Cflow and Optimizations . . . 40
3.3 State Effects Cflow . . . 42
Regenerating Cflow . . . 44
3.4 Related Work . . . 48
3.5 Summary . . . 49

II Static Semantics . . . 51

4 Abstracting Pointcuts and Advice to Effects . . . 53
4.1 Computational Effects . . . 54
4.2 Effect Analysis for PROC . . . 56
Effect Strings for Procedures . . . 56
Effect Strings for Dynamic Join Points . . . 59
Effect Strings for Pointcuts . . . 61
Pointcut Effect Reports . . . 62
Effect Strings for Advice Bodies . . . 64
Advice Effect Reports . . . 65
4.3 Exceptions and Threads . . . 68
4.4 Effect Analysis for AspectScheme . . . 70
4.5 Related Work . . . 71

4.6 Summary 72

5 Classifying Pointcut and Advice Interactions 75
5.1 Simple Interactions 76
Control Flow Categories 76
5.2 Data Interaction Categories 78
Input (Output) Interactions 82
Exception Interactions 84
Concurrency Interactions 85
Summary of Interactions 86
5.3 Compound Interactions 86
5.4 Example Interactions and Reports 88
Tracing 88
Move Limiting 89
Exception Logging 90
Runnable With Return 91
5.5 Other Analyses and Related Work 92
Clifton et al. 92
Katz et al. 93
Dantas et al. 94
5.6 Summary 95

6 Conclusion 97
6.1 Contributions 97
6.2 Open Questions 99
Extending and Formalizing 99
Object-Oriented Languages 101
An Effect Checking Tool 102

Works Cited 103

Appendices **117**

A AspectScheme CEKS Semantics **119**

A.1 Syntactic Categories 119

A.2 Transition Rules 122

B AspectScheme 2.3 Implementation **127**

C PROC Implementation **137**

C.1 Syntax 138

C.2 Parser 139

C.3 Elaborator 141

C.4 Evaluator 143

C.5 AOP Constructs 149

C.6 Environments 152

C.7 Top Level 153

Subject Index *155*

Citation Index *157*

Tables

1	Dynamic Join Points	60
2	Dynamic Join Point Shadows	61
3	Control Flow Interactions	78
4	Data Interactions	82
5	Input (Output) Interactions	83

Figures

1	PROC Abstract Syntax	8
2	PROC Big-step (Direct) Semantics	9
3	PROC Shorthand Expressions	10
4	PROC Small-step (CPS) Semantics — Continuations	11
5	PROC Small-step (CPS) Semantics — Evaluator	13
6	PROC Small-step (CPS) Semantics — Primitives	14
7	PROC Pointcuts — Abstract Syntax	16
8	PROC Pointcuts — Implementation	17
9	PROC Advice Declaration – Abstract Syntax	18
10	PROC Before and After Advice	18
11	PROC Advice – Elaboration and Matching	20
12	PROC Advice – Frames	21
13	PROC Advice – Weaving	22
14	PROC Advice – Invocation	23
15	PROC Advice – Proceed	23
16	CEKS Primitives Transitions	31
17	CEKS Primitive Operations	32
18	CEKS Around Transition Rules	33
19	CEKS Equality Operation	35
20	CEKS Primitive Application Transitions	36
21	CEKS Application Transitions	37
22	CEKS Continuation Marking Application Rules	39
23	CEKS Dynamic Join Point Construction 1	39
24	CEKS Advice Weaving	40
25	CEKS Cflow Pointcut Implementation	41
26	Tail-recursive Factorial and Advice	42
27	CEKS Application Rules 2	43
28	CEKS Dynamic Join Point Construction 2	43
29	CEKS Advice Weaving 2	44
30	Cflow Model Advice	45
31	Invalid Cflow Translation	45
32	Valid Cflow Translation	47

33 Expression Codewalker 57
34 Shadow Frames 57
35 Unfolding Expressions into CPS Form 58
36 CPS Intermediate Language Effects 60
37 Inferred Pointcut Effects 62
38 Proceed Counting 66
39 Expressions for Exceptions and Threads 68
40 Orthogonal Interaction 79
41 Independent Interaction 80
42 Observation Interaction 80
43 Actuation Interaction 81
44 Influence Interaction 81
45 Interference Interaction 82
46 Multiple Advice at Dynamic Join Points 87
47 Tracing Instrumentation 88
48 Move Limiting 89
49 Exception Logging Instrumentation 90
50 Runnable With Return 91
51 Object-Oriented Dynamic Join Points 101

Preface

Mathematics holds elegance in high regard. From my undergraduate studies in pure mathematics, I developed an appreciation that a simple, clear explanation of existing results is not only a legitimate scientific advance in itself, but also the engine that moves the discipline forward. With clarity comes insight — and applicability.

This dissertation aims to provide a simple, clear presentation of dynamic join points, pointcuts, and advice, based on fundamental principles in programming languages theory. The highest accolade this work can garner was provided by Varmo Vene during the summer of 2005, when he said to me: "Now I understand what aspects are about". If you, the reader, come to the conclusion of this dissertation and say "but that's obvious", then I will have succeeded in my endeavour.

From this clarity, will come two important insights. First, that pointcuts and advice are natural for any dynamic semantics. Second, layering these kinds of aspects is a challenging task; there is no single correct interaction. Programmers must spell out their intentions, but we can help to highlight contentious places.

This dissertation is certainly not the last word on this subject. Indeed, this presentation hints at a deeper understanding of the modularity offered by pointcuts and advice. Just as classes classify class-instances (objects) which abstract primitive values, there is a tantalizing parallel where dynamic aspects classify aspect-instances which abstract continuation structure. Filinski's categorical duality between continuations and values may connect the two hierarchies together, giving us a more unified theory of modularity over data and control.

As Gregor Kiczales once said, "the fun has just begun..."[92].

CHRIS DUTCHYN
Vancouver, Canada
November 6, 2006

Acknowledgments

Portions of Chapter 2 originally appeared in Wand et al. [167, 168, 169]. Portions of Section 3.1 were published as Dutchyn et al. [58]. Portions of section 3.2 appeared first in Masuhara et al. [113, 114].

Although only one name appears on the cover, this is not an individual effort. Many people played large and small rôles in getting me here today:

1. Duane Szafron, for unflagging confidence from beginning to end, supplemented by regular doses of candor;
2. Kris de Volder and Hidehiko Masuhara for listening to my unsupported rantings about programming languages, and setting me straight afterwards;
3. Norm Hutchinson, for stepping in for Gregor during his leave and helping me connect with the right research community;
4. Tim Sheard, for guidance during my time of directionless drifting;
5. Richard Gabriel, for *Patterns of Software* — the impulse that got me back to grad school;
6. Jonathan Sillito, Brian de Alwis, Andrew Eisenberg, and the other SPL lab members, for lively discourse and papers;
7. Yvonne Coady and Jan Hanneman, for going first and showing me how to graduate;
8. Holly Kwan, for good cheer no matter how dark the day seemed;

9. Will Evans and Alan Hu, for reading my error-ridden drafts and asking tough questions;
10. Gail Murphy, for advising me, even though I wasn't your student;
11. Dan Friedman, for the inspirations of EOPL, `Brown`, and reflective monads;
12. Shriram Krishnamurthi, for inviting me into an enviable project; and, along with Kathi Fisler, for advice in getting my career going;
13. Mitch Wand, for showing me how to read, understand, and write "Greek" in the sense of Mattias Felleisen;
14. Last and most important, Gregor Kiczales, for ideas, words, tolerance, criticism, and support over the long haul — your clarity and incisiveness in writing, teaching, and thinking are a model for me.

As always, any errors that remain are mine alone.

Dedication

for Mom, who never saw this completed…

Chapter 1

Introduction

Current programming languages offer many ways of organizing code into conceptual blocks, whether 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. Recent work [31; 52; 109; 152] 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.

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

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.

programmer cannot simply add or delete these features from a program, since they are not separable units.

1.1 Motivation

Recently, many researchers have proposed aspect-oriented software development as a method for organizing crosscutting features [13; 26; 79; 94; 107; 124; 157]. In particular, Kiczales et al. [94] have presented aspect-oriented programming (AOP); in this paradigm, the fragments of any given crosscutting feature precipitate into a separate component, called an *aspect*. In addition to containing the code necessary for a feature, the aspect must indicate how this code should combine with other modules to provide the desired behavior.

Kiczales et al. also implemented a practical aspect-oriented extension to Java, called AspectJ, which allows the programmer to define aspects [93]. One portion of this implemented language provides dynamic aspects in the form of pointcuts and advice. Simplistically, pointcuts identify dynamic join points — places where features interact, and advice implement a feature relative to the join point.

Our task is to provide a formal model for these constructs: dynamic join points, pointcuts, and advice; and, to explore how features implemented in this way might interact.

Thesis

We claim that a model of dynamic join points, pointcuts, and advice based on a continuation-passing style interpreter provides a fundamental account of these AOP *mechanisms; and that this account frames interesting type-and-effect properties of the mechanisms, such as the range of interactions between advice and advised dynamic join points, and provides a general framework for describing these interactions.*

Contributions

This research provides two semantic descriptions of dynamic join points, pointcuts, and advice for procedural languages.

The first semantic specification, a dynamic semantics, moves from our previously published expression-oriented, big-step system to a novel continuation-

based, small-step semantics. This translation yields an elegant model of dynamic join points as principled program control points, pointcuts as identifiers of these points, and advice as specializers of the behaviour of these control points. The second semantic specification, a static semantics, captures the essential abstraction of continuations, that of computational effects, and develops an abstraction of pointcuts and advice with regard to the effects they express. This abstraction to effects supports and refines existing aspect classifications, yielding interesting types-and-effects properties for dynamic joinpoints, pointcuts, and advice.

The specific contributions are:

1. A novel development of continuation-based dynamic semantics for dynamic join points, pointcuts, and advice for a first-order, mutually-recursive procedural language showing that
    a) Dynamic join points, pointcuts, and advice aspects can be modeled directly in continuation semantics; without the need for labels or continuation marks,
    b) Principled dynamic join points arise naturally, as continuation frames, from describing programming languages in continuation semantics, and
    c) Advice acts as a procedure on these continuation frames, providing specialized behaviour for them.
2. An application of this construction to a higher-order procedural language, Scheme, yielding a semantic description of AspectScheme which includes lexically-scoped and dynamically-scoped pointcuts and advice.
3. An implementation of AspectScheme, constructed as a language extension to PLT Scheme [72], using macros and their language extension points, with lexically-scoped and dynamically-scoped aspects, as well as the more usual top-level (declarative) pointcuts and advice aspects.
4. A demonstration that *cflow* pointcuts, which identify dynamic join points based on control flow context, break tail-call properties of programming languages and add a state effect into the languages.
5. A static semantics that focuses on the key property of continuations: that they carry computational effects. We

   a) characterize dynamic join point shadows by their input, output, state access and state mutation regions;

   b) associate dynamic join point effects with pointcuts, yielding reports summarizing and contrasting these effects;

   c) characterize advice bodies, describing their input, output and state effects as well as repetition of join point behaviour.

6. An effect reporting algorithm that extends ones already accepted for aspect-oriented languages. Ours includes

   a) five control interaction classes that cover a broader range than the existing classes,

   b) six data interactions, including one missing from the existing analyses,

   c) an alternate input (output) categorization with four categories

   d) exception categorization that helped highlight an AspectJ/Java inconsistency, and

   e) three simple concurrency interactions.

1.2 Plan of Presentation

This work proceeds in two major parts. The first part deals with dynamic semantics. In Chapter 2 we introduce direct semantics for Proc, an eager, call-by-value, first-order recursive procedural language. We then pass to the continuation-passing style (cps) semantics, and characterize dynamic join points, pointcuts, and advice in terms of these continuation structures. To emphasize the generality of our construction, we provide a second account in Chapter 3 where we identify dynamic join points, pointcuts, and advice in a core implementation of Scheme, a higher-order procedural language.

The second part deals with static semantics. In Chapter 4 we present computational effects and show how to abstract effect descriptions for our pointcuts and advice. In Chapter 5, we demonstrate the utility of these effect abstractions by classifying them in a taxonomy that extends the existing formulations for reasoning about dynamic aspects. Last, in Chapter 6, we summarize the work and consider some avenues for additional research.

Part I

Dynamic Semantics

Chapter 2

A Model for Dynamic Join Points, Pointcuts, and Advice

principle /'prɪnsɪp(ə)l/ *n.*

1. a fundamental truth or law as the basis of reasoning or action.
2. a) a personal code of conduct.
   b) (in *pl.*) such rules of conduct.
3. a general law in physics etc.
4. a law of nature forming the basis for the construction or working of a machine etc.
5. a fundamental source; a primary element.

principled /'prɪnsɪp(ə)ld/ *adj.* based on or having (esp. praiseworthy) principles of behaviour.

The Concise Oxford Dictionary, 8ed. [6]

This chapter comprises the core of our dissertation. 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.

```
(define-struct PROG (decls body))    ;; <(id * decl)...* exp>

(define-struct LIT (val))            ;; literal value
(define-struct VAR (id))             ;; variable
(define-struct IFX (test then else)) ;; conditional <exp * exp * exp>
(define-struct APP (id rands))       ;; application <id * exp...>

(define-struct PROC (ids body)) ;; <id...* exp>
;;; primitives are procedures
```

Figure 1: Proc Abstract Syntax

2.1 A Procedural Language – Direct Semantics

As with other semantics presentations [56; 57; 169], we choose to work with a first-order, mutually recursive procedural language, Proc. In the next chapter, we will re-examine this construction for a core higher-order procedural language. Throughout this chapter, our systems are given as definitional interpreters, as introduced by Reynolds [135, 137], in the style of Friedman et al. [74]. This interpreter-based approach to modeling various AOP mechanisms originated with our work in the Aspect Sandbox [56; 57] and related papers [113; 114; 167; 168; 169], and was later adopted by others, including Filman [69].

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 [91].

We should point out that several usual constructs are lacking from our language; but this does not impair its expressiveness. In particular, the usual constructs are

- (*SEQ* $x_1 \ldots$) 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.

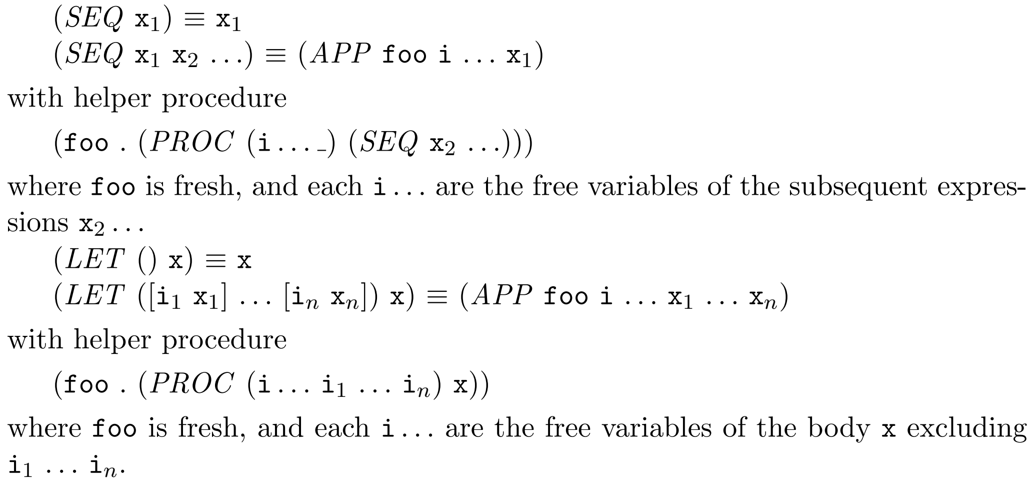
```
;;; values :: integers, booleans
;;; evaluator
(define (eval x r)
  (cond [(LIT? x) (LIT-val x)]
        [(VAR? x) (lookup r (VAR-id x))]
        [(IFX? x) (let ([v (eval (IFX-test x) r)])
                    (eval ((if v IFX-then IFX-else) x) r))]
        [(APP? x) (let ([vs (map (λ (x) (eval x r)) (APP-rands x))])
                    (let ([p (lookup-proc/prim (APP-id x))])
                      (cond [(PROC? p) (eval (PROC-body p)
                                             (bind (PROC-ids p) vs empty))]
                            [(procedure? p) (p vs)])))]))
(define *procs* `([+ . ,(λ (vs) (+ (car vs) (cadr vs)))]
                  [display . ,(λ (vs) (display (car vs)) 0)]
                  [newline . ,(λ (vs) (newline) 0)]))
(define (run s)
  (let ([g (parse-prog s)])
    (set! *procs* (cons (PROG-decls g) *procs*))
    (eval (PROG-body g) empty)))
```

Figure 2: Proc Big-step (Direct) Semantics

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

2.2 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 Landin [102] and Strachey [155] to model control flow in programs. Later, Reynolds [136] applied them to ensure that semantics given by definitional interpreters yields a formal model independent of the defining language control constructs.



$(SEQ\ \mathtt{x}_1) \equiv \mathtt{x}_1$
$(SEQ\ \mathtt{x}_1\ \mathtt{x}_2\ \ldots) \equiv (APP\ \mathtt{foo}\ \mathtt{i}\ \ldots\ \mathtt{x}_1)$

with helper procedure

$(\mathtt{foo}\ .\ (PROC\ (\mathtt{i} \ldots _)\ (SEQ\ \mathtt{x}_2\ \ldots)))$

where `foo` is fresh, and each `i`... are the free variables of the subsequent expressions $\mathtt{x}_2 \ldots$

$(LET\ ()\ \mathtt{x}) \equiv \mathtt{x}$
$(LET\ ([\mathtt{i}_1\ \mathtt{x}_1]\ \ldots\ [\mathtt{i}_n\ \mathtt{x}_n])\ \mathtt{x}) \equiv (APP\ \mathtt{foo}\ \mathtt{i}\ \ldots\ \mathtt{x}_1\ \ldots\ \mathtt{x}_n)$

with helper procedure

$(\mathtt{foo}\ .\ (PROC\ (\mathtt{i} \ldots\ \mathtt{i}_1\ \ldots\ \mathtt{i}_n)\ \mathtt{x}))$

where `foo` is fresh, and each `i`... are the free variables of the body `x` excluding $\mathtt{i}_1\ \ldots\ \mathtt{i}_n$.

Figure 3: Proc Shorthand Expressions

The CPS transformation [48] of our interpreter is systematic, following closely that of Hatcliff and Danvy [82]. In essence, we treat each of the **let** expressions in the direct *eval* semantics as a *monadic let* [119; 120]. These **let**s 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 [45], but Ager et al. [5] provide a systematic defunctionalization of these closures into tagged structures and an *apply* procedure that gathers the operations of each closure. 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, *ARG* and *CONS*, to support multiple arguments to procedures[2]. These two continuations provide a strict right-to-left evaluation order for procedure operands. This choice is arbitrary. We could have supplied a non-deterministic ordering in the CPS semantics, but that would distract us from our focus. The essential notion is that these supporting continuations have no basis in the direct semantics:

2. We refer to continuations which arise from the CPS transformation and provide sequencing which is unspecified in the big-step semantics as *auxiliary*.

```
;;; frames
(define-struct TEST (env then else)) ;; < env * exp * exp> ⊢ ¬boolean
(define-struct CALL (id)) ;;<id> ⊢ ¬val...
(define-struct EXEC (args)) ;; <val...> ⊢ ¬proc
(define-struct ARG (env exp)) ;;<env * exp> ⊢ ¬val...
(define-struct CONS (vals)) ;; <val...> ⊢ ¬val

;;; continuations :: frm...
(define (push f k)
  (cons f k))

(define ((pop end step) k)
  (if (null? k)
      (end)
      (step (car k) (cdr k))))
```

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

they serve only to bridge the gap between the big-step and small-step systems.

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 *CONS* 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 [159] provides the details.

For our restricted procedural language, the full power of the λ-calculus is not required. Indeed, in the λ-calculus, the *TEST* continuation is unnecessary as well. A simple syntactic transformation makes the consequent clauses into thunks (parameterless closures [47]). True and False become binary procedures that simply apply one or the other thunk. In summary, we characterize *ARG*, *CONS*, and *TEST* as *auxiliary* continuations[3].

3. These should not be confused with *serious* and *trivial* continuations [135; 137], nor with *administrative* continuations [73]. *Serious* continuations are ones which may not terminate, i.e. they may lead to recursion or other indefinite stack growth. Procedure application continuations are serious. *Trivial* continuations are ones which perform some trivial (i.e. known to terminate) operation on the values at hand, and then call their continuation. *Administrative*

The defunctionalized CPS definition of our interpreter is given in Figure 5.

Our construction is standard, except in three respects. First, we extend Agere's construction to explicitly linearize the continuation. In Agere'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 * frm... → frm...* — extends an existing continuation with another frame.
- *pop :: ((→ unit) * ((frm * frm...) → unit)) → frm... → unit* — takes a continuation, and either
  - applies the first procedure (*end*) because the continuation is empty, or
  - applies the second procedure (*step*) to the top continuation frame and the rest of the continuation.

In our case, the halt continuation is represented by the empty list.

Second, 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** [61; 63], as primitives. This is seen in Figure 6.

Third, our implementation distinguishes the lookup of procedures into a separate continuation, *EXEC*. Ordinarily, we would require only one continuation, *CALL*, to await the evaluation of the operands into argument values. That single continuation would be responsible for locating the desired 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** expressions present in the case of an *APP* expression. Other one-step [50] and

continuations are those which can be automatically reduced during the CPS transformation. For example, translating a **let** into a procedure application yields an administrative continuation where the closure is immediately applied; the continuation that performs the application can be administratively reduced. This leads to A-normal forms, the subject of Flanagan et al. [73].

```
;;; values :: integers, booleans
;;; evaluator
(define (eval x r k)
  (cond [(LIT? x) (apply (LIT-val x) k)]
        [(VAR? x) (apply (lookup r (VAR-id x)) k)]
        [(IFX? x) (eval (IFX-test x)
                        r
                        (push (make-TEST r IFX-then IFX-else) k))]
        [(APP? x) (evlis (APP-rands x)
                         r
                         (push (make-CALL (APP-id x)) k))]))

(define (evlis xs r k)
  (if (null? xs)
      (apply '() k)
      (evlis (cdr xs)
             r
             (push (make-ARG r (car xs)) k))))

(define ((step/prim v) f k)
  (cond [(TEST? f) (eval ((if v TEST-then TEST-else) f)
                         (TEST-env f)
                         k)]
        [(CALL? f) (apply (lookup-proc/prim (CALL-id x))
                          (push (make-EXEC v) k))]
        [(EXEC? f) (cond [(PROC? v) (eval (PROC-body v)
                                          (bind (PROC-ids v)
                                                (EXEC-args f)
                                                empty)
                                          k)]
                         ;; primitives now take args and cont
                         [(procedure? v) (apply (v (EXEC-args f) k))])]
        [(ARG? f) (eval (ARG-exp f) (ARG-env f)
                        (push (make-CONS v) k))]
        [(CONS? f) (apply (cons v (CONS-vals f)) k)]))

(define (halt v)
  (display v) (newline))

(define (apply/prim v k)
  ((pop (halt v)
        (step/prim v))
   k))

(define apply apply/prim)
```

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

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

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

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

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

A-normal [73; 140] transformations optimize portions of this transformation, usually the inner **let**. 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.

2.3 Exposing Our AOP Constructs

With these preliminaries, we are prepared to expose dynamic join points in Proc, and provide syntax to denote pointcuts and advice. We need to describe three items:

1. **dynamic join points** — "principled points in the execution" [94]. These will be states in the interpreter where values are applied to non-auxiliary continuation frames.

2. **pointcuts** — "a means of identifying join points"[4]. 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"[5]. This is implemented as the advice body as a procedure applied to the continuation frame.

4. ibid
5. ibid

Dynamic Join Points

Dynamic join points are the first abstraction in our model. Other semantic models provide a list of dynamic join points separately. For us, join points are activations of certain continuation frames. Recall that we introduced additional auxiliary frames to support our eager, left-to-right evaluation order in the CPS semantics. Therefore, we adopt the following principle:

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

Therefore, we have two frames corresponding to dynamic join points:

- $\mathit{CALL}\ (\mathit{id} \vdash \neg \mathit{val}\ldots)$ — consumes a list of values and emits an *EXEC* continuation frame with the identifier of the procedure to execute,
- $\mathit{EXEC}\ (\mathit{val}\ldots \vdash \neg \mathit{proc})$ — consumes a procedure and evaluates its body with identifiers bound to the stored values.

The type signatures indicate the type of the information stored in the continuation frame before the turnstile ($\vdash$) and the type of the expected value to be consumed by the continuation after the $\neg$. Jouvelot and Gifford [86] and Murthy [122] originated the use of $\neg$ types; Thielecke [159] explores this in detail.

Therefore, our dynamic join points are simply the activation of either of these continuation frames. In each case, a dynamic join point has two items available:

1. a procedure, either by name (in the case of *CALL*) or as an actual structure (in the case of *EXEC*),
2. a list of values corresponding to the arguments to the procedure.

.

In our model, dynamic join points make accessible the latent control structure of the language semantics. Dynamic join points correspond to continuation frames, and are modeled by states within the interpreter. If the semantics of the language change, through the addition of new constructs or a change in the semantic equations such as explicitly partially-evaluating some terms, the set of dynamic join points would be expected to change.

Our dynamic join points systematically align with points in the model that are well-accepted as being semantically meaningful. Our principle defines this

```
;;; natural pointcuts
(define-struct CALLPC (pname anames)) ;<id id...>
(define-struct EXECPC (pname anames)) ;<id id...>
;;; combinational pointcut
(define-struct ORPC (pc1 pc2)) ;<pc pc>
```

Figure 7: Proc Pointcuts — Abstract Syntax

systematic alignment. In other models, some have framed dynamic join points as program rewrite points [11; 139]. Other accounts have dynamic join points appear as an ad-hoc list, including in our earlier work [169].

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 *call* dynamic join points, and another for identifying *exec* dynamic join points. We adopt the following syntax for pointcuts. It contains two 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 8.

In the case of a *CALLPC* pointcut, we ensure that the frame is a *CALL* frame, and that it holds a procedure name equal to the one given in the pointcut. For a *EXECPC* pointcut, we ensure that the frame is an *EXEC* frame and that the supplied value is a procedure whose name is equal to the one given in the pointcut.

We also include one combinational pointcut. It is *ORPC*, 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 *display*`X` procedures, each with a different output format, to be a single display concern.

```
;;; pointcut matching – returns #f or ids
(define ((match-pc p) v f)
  (cond [(CALLPC? p) (and (CALL? f)
                          (string=? (CALLPC-pname p)
                                    (CALL-id f))
                          (CALLPC-ids p))]
        [(EXECPC? p) (and (EXEC? f)
                          (eq? v (lookup/proc (EXECPC-pname p)))
                          (EXECPC-ids p))]
        [(ORPC? p) (or ((match-pc (ORPC-p1 p)) v f)
                       ((match-pc (ORPC-p2 p)) v f)))))
```

Figure 8: Proc Pointcuts — Implementation

This combinational pointcut provides a simple specialization ordering to pointcuts; and, by extension, advice. Any given pointcut, `A`, is more specialized than *ORPC*(`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.

If 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 an *ORPC* pointcut yields the identifiers for the matching sub-pointcut. This means that each sub-pointcut must provide the same identifiers.

One important property of pointcuts is that they *identify* dynamic join points, but *do not alter* their semantic behaviour. It provides a clear distinction among the rôles assigned to each construct. This matches our intuition about pointcuts. For example, consider the case of advice which simply proceeds; there should be no effect regardless of what the pointcut attached to the advice body. The effect of a piece of advice should be determined by the advice body, not the pointcut.

Principle. *Pointcuts identify dynamic join points; they do not do not alter the semantic behaviour of the identified dynamic join points.*

This will have repercussions when we consider the *cflow* pointcut found in AspectJ and other languages. That pointcut is responsible for identifying dynamic join points that occur during the control flow of another dynamic join point.

(**define-struct** *ADVICE* (*pc body*)) ;; <pointcut expression>

Figure 9: PROC Advice Declaration – Abstract Syntax

(*BEFORE* `pc x`) ≡ (*AROUND* `pc` (*SEQ* `x proceed`))

(*AFTER* `pc x`) ≡ (*AROUND* `pc` (*APP* `foo (proceed)`))

with fresh helper procedure

(`foo .` (*PROC* (*v*) (*SEQ* `x` *v*)))

Figure 10: PROC Before and After Advice

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. However, dynamic join points are 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.

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 9. Advice are declarations in our model, just like procedures. Therefore, they will have identifiers bound to them, just as procedures do.

In our system, all advice in our system 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 two possible orderings of the advice body and *proceed*, as shown in Figure 10.

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 x) (if x 1 2))

(+ (pick #t) 3)
```

is applied to #t to yield a new value 1. Filinski [65] first recognized that *pick* transforms the continuation of the procedure application from

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

to

```
(λ (b)   ; await boolean
  (let ([n (if b 1 2)])   ; select number
    ((λ (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 [86] 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.

In Filinski's *symmetric lambda calculus* [65], 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 Figure 11.

```
;;; advice elaboration
(define *advice* #f)

(define (run s)
  (let ([g (parse-prog s)])
    (set! *procs* (cons (collect (λ (d)
                                   (PROC? (cdr d)))
                                 (PROG-decls g))
                        *procs*))
    ; collect advice declarations
    (set! *adv* (collect (λ (d)
                           (ADVICE? (cdr d)))
                         (PROG-decls g)))
    (eval (PROG-body g) empty)))

;;; advice matching
(define-struct MATCH (ids adv))

(define (collect-matches v f)
  (collect (λ (a)
             (let ([ids (match-pc (ADVICE-pc a) v f)])
               (if ids
                   (make-MATCH ids a)
                   #f)))
           *advice*))
```

Figure 11: Proc Advice – Elaboration and Matching

Matching is also shown in Figure 11. We simply walk the elaborated list of advice, comparing the pointcuts and returning a *MATCH* containing the pointcut-match identifiers and the advice itself.

We need to have a new continuation frame to support advice execution. We also will end up needing a special marker frame to disable matching in the case that we have *proceed*ed to the actual dynamic join point itself. These frames are called *ADVEXEC* and *APPPRIM*, shown in Figure 12. The extensions to *step/prim* are shown in that figure as well. For simplicity, we make the final element of the matches be a representation of the original dynamic join point: for a *CALL* dynamic join point we store the procedure name, for an *EXEC* dynamic join point we store the *PROC* or procedure.

```
;;; advice continuation frames
(define-struct ADVEXEC (matches)) ;<match... | id | PROC | procedure>
(define-struct APPPRIM ()) ;indicates call step/prim rather than step – administrative

;;; step/prim contains original step
(define ((step/prim v) f k)
  (cond ;...;original content unchanged
        [(ADVEXEC? f) (invoke/adv (ADVEXEC-matches f) v k)]
        [(APPPRIM? f) (apply/prim v k)]))
```

Figure 12: Proc Advice – Frames

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. We need to implement a weaver that determines applicable advice at each dynamic join point. We implement *step/weave* and use it to re-define *apply* in our system. The implementation is shown in Figure 13.

Execution of advice is displayed in Figure 14. This figure shows that advice invocation parallels that of procedure execution.

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 15.

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[6]. It contains the remaining advice, if any, and the original procedure name (in the case of a *CALL* dynamic join point), the original *PROC* or procedure (in the case of an *EXEC* dynamic join point).

Recalling our principle that dynamic join points correspond to frame activations, we recognize that our new frame, *ADVEXEC* defines a new set of dynamic join points that may be matched against. By construction, all of our declara-

6. Note that the shorthand expressions (Figure 3) must carry this additional variable into helper procedures if they contain *proceed*. This is a trivial operation.

```
;;;step/weave weaves advice based on matches
(define ((step/weave v) f k) ;; step with advice weaving
  (let ([ms (collect-matches v f)])
    (if (null? ms)
        ((step/prim v) f k)
        (invoke-adv (append ms ;more advice
                            (cond [(CALL? f) (CALL-id f)] ;final proceed
                                  [(EXEC? f) v]
                                  [(ADVEXEC? f) (ADVEXEC-matches f)]))
                    (cond [(CALL? f) v] ; arguments to advice
                          [(EXEC? f) (EXEC-args f)]
                          [(ADVEXEC? f) v])
                    k))))

(define (apply/weave v k)
  ((pop (halt v)
        (step/weave v))
   k))

(define apply apply/weave)
```

Figure 13: Proc Advice – Weaving

tions are bound to identifiers, advice declarations will also have names. Hence, we can easily provide an advice-execution dynamic join point, and its associated matching operation. By construction, all activations of *ADVEXEC* frames are processed by *step/weave*, so the weaving of additional behaviour is automatic. We simply need to have *invoke/adv* recognize that if the last element in the match is a list, then we are back to proceeding the original advice. The call structure that makes this so is:

- *apply* calls *step/weave*
- *step/weave* looks for matching advice
  - if there is none, *step/prim* provides the fundamental behaviour of the dynamic join point
  - if there is a match, *invoke/adv* is called to evaluate arguments and push an advice execution dynamic join point

```
;;; advice invocation
(define (invoke/adv ms vs k)
  (let ([m (car ms)])
    (cond [(symbol? m) (apply vs          ;proceed to CALL
                              (push (make-APPPRIM)
                                    (push (make-CALL m)
                                          k)))]
          [(PROC? m) (apply vs            ;proceed to EXEC PROC
                            (push (make-APPPRIM)
                                  (push (make-EXEC m)
                                        k)))]
          [(procedure? m) (apply vs       ;proceed to EXEC prim
                                 (push (make-APPPRIM)
                                       (push (make-EXEC m)
                                             k)))]
          [(list? m) (apply vs            ;proceed to ADVEXEC
                            (push (make-APPPRIM)
                                  (push (make-ADVEXEC m)
                                        k)))]
          [(MATCH? m) (eval (ADVICE-body (MATCH-adv m))
                            (bind (cons '%proceed (MATCH-ids m))
                                  (cons (cdr ms vals))
                                  empty)
                            k)]
```

Figure 14: Proc Advice – Invocation

```
;;; proceed expression
(define-struct PROCEED (rands)) ;<exp...>

(define (eval x r k)
  (cond ;...;original cases unchanged
        [(PROCEED? x) (evlis (PROCEED-rands x)
                             r
                             (push (make-ADVEXEC (lookup '%proceed r))
                                   k))]))
```

Figure 15: Proc Advice – Proceed

- *proceed* expressions call *invoke/adv* 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 pointcut associated with that advice. 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.

2.4 Comparison to Other Semantics

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

Aspect Sandbox

In joint work, Dutchyn et al. [56, 57] and Wand et al. [167], we developed a number of semantic models of aspect-oriented programs, both for object-oriented and procedural languages. The culmination of that work, Wand et al. [169] 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. [169] 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 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 [93]. The 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. We also avoid *within*, a pointcut that identifies dynamic join points based on lexical structure. In our

model, *&&* provides no additional expressive power, so we do not include it. In summary, our pointcut language provides a reasonable fit for our model approach.

Another sort of pointcut is also part of the Aspect Sandbox: *cflow*. In some sense, it is the dynamic equivalent of *within*: it identifies dynamic join points which are within the control flow context of another dynamic join point. As this pointcut matches dynamic join points based on their dynamic context, it has subtle interaction with tail-call optimization. Furthermore, because it motivates the second part of the dissertation on effects, we defer its discussion until Section 3.2.

Wand et al. [169] chooses to statically weave execution dynamic join points into the procedure declarations. In the case of multiple *proceed* calls, this can lead to code bloat, as the body of the procedure is replicated. They also need to ensure freshness of variables, as inserting a **let** around a *proceed* can cause accidental capture. Our implementation avoids this duplication of code and maintains existing lexical structures.

AspectScheme

Our model provides a straightforward way to implement dynamic join points, pointcuts, and advice languages. We attempted this with Scheme, yielding AspectScheme. A semantic description of AspectScheme, based on our principle of dynamic join points as continuation frames, is given in the next chapter.

This author contributed the semantic description of AspectScheme [58] and the online implementation [55]. AspectScheme models join points as procedure applications in the 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 a 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. [5]'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 *EXEC* dynamic join point, but with additional context. However, because dynamic join points are first-class objects, namely 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.

The published AspectScheme language provides some other pointcuts, one of these is *cflow*(pc). This walks the context of pending procedures to see if the desired one is encountered. Unfortunately, to support the expected semantics, general tail-call optimization in Scheme must be discarded, as we will see in the next chapter.

PolyAML and μABC

Dantas et al. [41] provide 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 [70] aspects, which must be *before* and *after* only. A later paper [42] solves the typing difficulties with *around* advice, using novel local type inference techniques.

Their monomorphic machine is described in terms of context semantics [4]. 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. [20] provides an untyped core calculus for aspects. As Dantas et al. [41] 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.

Other Related Work

Several other semantic formulations for aspects have been offered.

Douence et al. [53] 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 [8] provides a process-calculus description of aspects. Oblivious aspects are provided. Constrained by encapsulated processes, full *around* aspects are not possible.

2.5 Summary

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 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.

Chapter 3

Advice in Higher Order Languages

In this chapter, we apply our framework to Scheme, a higher-order language, making pointcuts and advice accessible to the programmer. We present the dynamic semantics of this language, AspectScheme, in terms of a CEKS machine. This framework will enable us to make precise statements about properties of dynamic join point, pointcut, and advice languages. In particular, we will investigate the *cflow* pointcut, including two different potential formal models. From this, we will recognize state, the computational effect inherent in this pointcut.

3.1 AspectScheme Model

The AspectScheme programming language is described in Dutchyn et al. [58]; the dynamic semantics given here draw heavily on that published presentation. One focus of the published presentation, scoping of dynamic join points, pointcuts, and advice aspects [54; 55], is not examined in detail here; but informs our understanding of the interactions of these kinds of AOP.

We begin by laying out the semantics for a functional language which expresses dynamic join points, pointcuts, and advice. We first give a primer on the machine model which we use to define our operational semantics. Subsequent sections then explain some key rules, including those for declaring these constructs, testing function equality, and applying functions. Last, we informally sketch the connection between the implementation, given in Appendix B and the

operational semantics.

Background on the CEKS machine

We use a variation on the CEKS machine [62] as the model for our semantics.

We have three reasons for using the CEKS machine in defining our semantics.

1. Pointcuts identify points in the computation; the structure of our defunctionalized continuations give us this concrete representation.

2. Because the machine uses an environment to maintain variables, we can easily add a second environment to keep track of aspects in scope.

3. Programmers often use side-effects in writing useful aspects (e.g. logging, tracing, error reporting); hence, we include a model that contains an abstract store.

The CEKS model defines program behavior by a transition relation from one program state to the next. Transitions that key on an expression correspond to *eval* clauses; and, transitions that key on a value correspond to *apply* clauses. We distinguish these transitions by differing arrows,

- $\triangleright$ for eval transitions and

- $\blacktriangleright$ for apply transitions.

The CEKS machine adds two more pieces of information to the state of a computation. First, it pairs each control string with an environment that maps variable names to locations in an abstract store. Second, each state has an abstract store that maps locations to value-environment pairs. Formally, we represent the state of a computation with a triple of the following form:

1. The control string (C) and its environment (E).

2. The current continuation (K).

3. The current store (S).

In order to use a CEKS machine, we must describe how to initiate a computation, and to recognize when it has terminated. Given a program, a closed

$$\langle\langle(o\ M_1\ \ldots M_n), E\rangle, K, S\rangle$$
$$\quad \triangleright_{prim}\ \langle\langle M_1, E\rangle, \langle \mathsf{op\text{-}k}\ o, \langle\rangle, \langle\langle M_2, E\rangle, \ldots, \langle M_n, E\rangle\rangle, K\rangle, S\rangle$$

$$\langle VC_m, \langle \mathsf{op\text{-}k}\ o, \langle VC_{m-1}, \ldots, VC_1\rangle, \langle MC_{m+1}, \ldots, MC_n\rangle, K\rangle, S\rangle$$
$$\quad \blacktriangleright_{prim}\ \langle MC_{m+1}, \langle \mathsf{op\text{-}k}\ o, \langle VC_m, \ldots, VC_1\rangle, \langle MC_{m+2}, \ldots, MC_n\rangle, K\rangle, S\rangle$$

$$\langle VC_n, \langle \mathsf{op\text{-}k}\ o, \langle VC_{n-1}, \ldots, VC_1\rangle, \langle\rangle, K\rangle, S\rangle\ \ \blacktriangleright_{prim}\ \langle \delta(o, VC_1, \ldots, VC_n), K, S\rangle$$

Figure 16: CEKS Primitives Transitions

top-level expression M, the machine is initialized with the triple:

$$\langle\langle M, E_0\rangle, \mathsf{mt\text{-}k}, S_0\rangle$$

where $E_0 \equiv x \mapsto$ error is the initial environment that binds no variables, and $S_0 \equiv \ell \mapsto$ error is the initial store that binds no locations. The machine steps through transitions until a terminal state $\langle\langle V, E\rangle, \mathsf{mt\text{-}k}, S\rangle$ is reached, whereupon V is the final value of the program.

During the execution of the CEKS machine, various primitive operations must be performed. In our case, we provide a minimal sufficient set for manipulating the list values we support: *empty?*, *cons*, *first*, and *rest*. The transition rules are given in Figure 16.

In this formal system, $VC = \langle V, E\rangle$ represents a closure of a value over an environment. Operands are evaluated left-to-right; with administrative continuation frame $\langle \mathsf{op\text{-}k}\, o, VC \ldots, MC \ldots, .\rangle$ The δ function receives the resulting value closures, and implements the actual primitives. Specifically, we define δ in Figure 17.

Declaring Advice

To declare advice, we use **around** and **fluid-around** expressions to expose the continuation structure of a functional programming language.

(**around** *pc adv body*)

$$\begin{aligned}
\delta(\mathit{empty?}, VC) &= \langle \mathsf{true}, E_0 \rangle \text{ if } VC = \langle \mathsf{empty}, E \rangle \\
&= \langle \mathsf{false}, E_0 \rangle \text{ otherwise} \\
\delta(\mathit{cons}, VC_1, VC_2) &= \langle (\mathsf{cons}\ \ VC_1\ \ VC_2), E_0 \rangle \\
\delta(\mathit{first}, \langle (\mathsf{cons}\ \ VC_1\ \ VC_2), E \rangle) &= VC_1 \\
\delta(\mathit{rest}, \langle (\mathsf{cons}\ \ VC_1\ \ VC_2), E \rangle) &= VC_2
\end{aligned}$$

Figure 17: CEKS Primitive Operations

(**fluid-around** *pc adv body*)

We will first describe the semantics of **around**; the semantics of **fluid-around** is nearly identical.

When the programmer declares an advice via **around**, the machine may later access the advice during function application. This situation resembles the use of variables: the programmer *declares* them with **lambda** or **let**, and later *accesses* them by variable references. Drawing on this analogy, we add a second environment to our machine—one for storing advice. The reduction rules for our model will be similar to those for the CEKS machine, except that closures now include both a variable environment and an advice environment. The template for a reduction rule now includes *advice environments*, A_i, in closures:

$$\langle\langle C_1, E_1, A_1\rangle, K_1, S_1\rangle \Rightarrow \langle\langle C_2, E_2, A_2\rangle, K_2, S_2\rangle$$

where $\langle C, E, A\rangle$ is either a value closure ($C = V$) , abbreviated as VC, or an expression closure ($C = M$) , abbreviated as MC. We provide $A_0 \equiv \emptyset$ as the initial, empty advice environment.

The evaluation of **around** has three reduction rules, given in Figure 18.

1. The first rule moves evaluation to the pointcut, M_{pc}, while remembering that the declaration was for a `static` advice.

2. The second rule says that once the pointcut computes to a value (VC_{pc}), evaluate the advice (MC_{adv}) next.

$$\langle\langle(\textbf{around}\ M_{\text{pc}}\ M_{\text{adv}}\ M), E, A\rangle, K, S\rangle$$
$$\quad \triangleright_{around}\ \langle\langle M_{\text{pc}}, E, A\rangle, \langle \textsf{around1-k}\ \texttt{static}, \langle M_{\text{adv}}, E, A\rangle, \langle M, E, A\rangle, K\rangle, S\rangle$$

$$\langle VC_{\text{pc}}, \langle \textsf{around1-k}\ \mathit{scope}, MC_{\text{adv}}, MC, K\rangle, S\rangle$$
$$\quad \blacktriangleright_{around}\ \langle MC_{\text{adv}}, \langle \textsf{around2-k}\ \mathit{scope}, VC_{\text{pc}}, MC, K\rangle, S\rangle$$

$$\langle VC_{\text{adv}}, \langle \textsf{around2-k}\ \mathit{scope}, VC_{\text{pc}}, \langle M, E, A\rangle, K\rangle, S\rangle$$
$$\quad \blacktriangleright_{around}\ \langle\langle M, E, A \cup \{\langle \mathit{scope}, VC_{\text{pc}}, VC_{\text{adv}}\rangle\}\rangle, K, S\rangle$$

$$\langle\langle(\textbf{fluid-around}\ M_{\text{pc}}\ M_{\text{adv}}\ M), E, A\rangle, K, S\rangle$$
$$\quad \triangleright_{fluid-around}\ \langle\langle M_{\text{pc}}, E, A\rangle, \langle \textsf{around1-k}\ \texttt{dynamic}, \langle M_{\text{adv}}, E, A\rangle, \langle M, E, A\rangle, K\rangle, S\rangle$$

Figure 18: CEKS Around Transition Rules

3. The third rule applies after both the pointcut and advice become values. The rule moves evaluation to the body of the **around** expression, but with an extended advice environment. We add the triple $\langle \mathit{scope}, VC_{\text{pc}}, VC_{\text{adv}}\rangle$ to the advice environment; that is, the scope tag (`static` for **around**), the pointcut value (VC_{pc}), and the advice value (VC_{adv}).

To support **fluid-around**, we simply add a rule similar to the first one for **around**, except that its scope tag is `dynamic`.

In short, the semantics of advice declaration say to evaluate the pointcut and advice, then add them (along with the appropriate *scope* tag) to the advice environment when evaluating the body. The continuation frames $\langle \textsf{around1-k}\mathit{scope}, VC, MC, \rangle$ and $\langle \textsf{around2-k}\ \mathit{scope}, VC, MC, \rangle$ are administrative.

Function Equality

Next we address the issue of function identity in a higher-order language. In AspectScheme, a pointcut can refer to one or more procedures; for example, the pointcut (*call open-file*) denotes dynamic join points representing calls to

the function *open-file*. Thus, at each function application, we must determine whether the function being applied is *open-file*. In a language like Java, this would be an easy test—we just use string equality to compare the name *open-file* with the name of the method being invoked. In a functional language, however, two problems arise. First, the term in the function position need not be a variable name—it may be an arbitrary expression that evaluates to a function. Second, even if the function *is* the variable *open-file*, we cannot tell by its name whether this was the *open-file* in scope when the advice was defined. Consider the following expression:

```
(let ([open-file (λ (f) ...)])
  (around (call open-file) trace-advice
    (let ([open-file (λ (f) ...)])
      (open-file "vancouver"))))
```

In this example, should the call to *open-file* invoke the advice body? The answer is no—because the *open-file* in the pointcut really refers to the outer *open-file*, while the function application refers to the inner *open-file*.

To cope with this challenge of function equality, we will borrow the definition of equality used in Scheme [91]. The predicate *eq?* in Scheme can be used to compare functions. One interpretation of function *eq?*-ness is:

> Two function closures are equal if they have the same textual source location and their environments are identical.

To capture this meaning, we assume that each **lambda** expression in the source program is labeled with a unique location identifier and each environment is labeled with a unique store location when it is constructed. In order to do this, we must extend our definition of environment to include a store location tag, $E :: \langle \ell, x \mapsto \ell \rangle$ with $E_0 = \langle \ell_0, x \mapsto \text{error} \rangle$, and store to include the set of locations allocated to environments, $S :: \langle \{\ell\}, \ell \mapsto VC \rangle$ with $S_0 = \langle \{\ell_0\}, \ell \mapsto \text{error} \rangle$ where ℓ_0 is initially allocated to E_0. This construction is similar to that given for R^5RS Scheme [91] in order to meet a minimal specification for *eq?*.

Two function closures are then *eq?* if and only if both location identifiers are the same and both environment locations are equal. An additional case for the δ function illustrates this definition, given in Figure 19.

This definition does not identify all functions that are observationally equal, but it is a conservative approximation of that relation that is both useful and can

$$\delta(eq?, \langle(\lambda\,(x)\;M)_t, \langle\ell, e\rangle, A\rangle, \langle(\lambda\,(x')\;M')_{t'}, \langle\ell', e'\rangle, A'\rangle) \\ = \langle \mathsf{true}, E_0, A_0\rangle \text{ if } t = t' \text{ and } \ell = \ell' \\ = \langle \mathsf{false}, E_0, A_0\rangle \text{ otherwise}$$

Figure 19: CEKS Equality Operation

be computed in constant time.

Primitive Function Application

Our language has two constructs for function application: the default application rule, which injects advice into the computation, and a "primitive" application (named **app/prim**), which does not observe advice. As we saw earlier, we use **app/prim** mainly to model AspectJ's **proceed** calls from within the body of an advice.

The semantics of **app/prim** are the same as that of application in the original CEKS machine, save for the question of how to handle the advice environment. With regular (variable) environments, we have two choices:

1. We can use *static scoping*—we evaluate the body of the procedure using the environment from its *definition site*.
2. We can use *dynamic scoping*—we evaluate the body of the procedure using the environment from its *application site*.

Since we support simultaneous static and dynamic advice, we use some advice from both advice environments. Specifically, we evaluate the body of the function using *static* advice from the site of definition, and *dynamic* advice from the site of application.

The evaluation of primitive application comprises three reduction rules, given in Figure 20.

1. The first rule moves evaluation to the function position, M_{fun}, and keeps track of the static advice from the aspect environment at the application site.

$$\langle\langle(\textbf{app/prim}\ M_{\text{fun}}\ M_{\text{arg}}), E, A\rangle, K, S\rangle$$
$$\quad \triangleright_{app/prim}\ \langle\langle M_{\text{fun}}, E, A\rangle, \langle \textsf{appprim1-k}\ \langle M_{\text{arg}}, E, A\rangle, A, K\rangle, S\rangle$$

$$\langle VC_{\text{fun}}, \langle \textsf{appprim1-k}\ MC_{\text{arg}}, A_{\text{app}}, K\rangle, S\rangle$$
$$\quad \blacktriangleright_{app/prim}\ \langle MC_{\text{arg}}, \langle \textsf{appprim2-k}\ VC_{\text{fun}}, A_{\text{app}}, K\rangle, S\rangle$$

$$\langle VC_{\text{arg}}, \langle \textsf{appprim2-k}\ \langle(\lambda\,(x)\ M)_t, E, A_{\text{fun}}\rangle, A_{\text{app}}, K\rangle, S\rangle$$
$$\quad \blacktriangleright_{app/prim}\ \langle\langle M, E', A'\rangle, K, S'\rangle$$

where

$$\begin{aligned} \langle E', S'\rangle &= \langle E, S\rangle + \{x \mapsto VC_{\text{arg}}\} \\ A' &= A_{\text{app}}|_{\texttt{dynamic}} \cup A_{\text{fun}}|_{\texttt{static}} \end{aligned}$$

Figure 20: CEKS Primitive Application Transitions

2. The second rule moves evaluation to the argument position, once the function is fully evaluated.

3. The third rule performs the actual application. It moves evaluation to the body of the **lambda** expression, extends the environment and store to reflect the parameter binding, and combines the two advice environments as described above.

To extend an environment and store with a variable and value, we use the following definition:

$$\langle\langle \ell, e\rangle, \langle L, s\rangle\rangle + \{x \mapsto VC\} \equiv \langle\langle \ell_e, e[x \mapsto \ell_v]\rangle, \langle L \cup \{\ell_e\}, s[\ell_v \mapsto VC]\rangle\rangle$$
$$\text{where } \ell_e, \ell_v \notin L \cup \text{dom}(S)$$

where $e[x \mapsto \ell]$ and $s[\ell \mapsto VC]$ employ the usual function extension.

These reduction rules for primitive application only differ from the original CEKS machine in one respect: they create the appropriate advice environment before evaluating the body of the function.

$$\langle\langle(M_{\text{fun}}\ M_{\text{arg}}), E, A\rangle, K, S\rangle$$
$$\quad \triangleright_{app}\ \langle\langle M_{\text{fun}}, E, A\rangle, \langle \mathsf{app1\text{-}k}\ \langle M_{\text{arg}}, E, A\rangle, E, A, K\rangle, S\rangle$$

$$\langle VC_{\text{fun}}, \langle \mathsf{app1\text{-}k}\ MC_{\text{arg}}, E_{\text{app}}, A_{\text{app}}, K\rangle, S\rangle$$
$$\quad \blacktriangleright_{app}\ \langle MC_{\text{arg}}, \langle \mathsf{app2\text{-}k}\ VC_{\text{fun}}, E_{\text{app}}, A_{\text{app}}, K\rangle, S\rangle$$

Figure 21: CEKS Application Transitions

Regular Function Application

Three transition rules dictate the evaluation of function application, given in Figure 21. The first two steps are standard, because we do not invoke advice until the function and its argument are evaluated. The first rule moves evaluation to the function position, remembering the advice environment from the application site. The second rule moves evaluation to the argument position. We now come to the heart of our semantics: the mechanism for invoking advice during function application.

Three things must happen during advice invocation. First, we must generate a dynamic join point representing the application; second, we must test and apply any advice transforming the dynamic join point; and third, we must allow the transformed dynamic join point to execute.

In our original publication of AspectScheme [58], dynamic join points were the application of a procedure to arguments. However, we also chose to adorn the dynamic join point with its context of pending procedure applications. In order to provide this context, we needed to provide a special *continuation mark* continuation frame to store each procedure while it was executing. This technique made *cflow*[7] pointcuts more direct, but made Scheme's tail-call optimizations unsound.

For many languages, tail-call optimization is permissible. For example, the Java programming language [78] explicitly supports tail optimization as discussed for the `StackOverflowException` class. The C and C++ programming languages

7. Recall that *cflow* pointcuts enable dynamic join points to be identified within the control-context of another dynamic join point.

specifications are silent about tail call optimization, and several popular implementations, including the GNU compiler suite include support for this feature. Therefore, presuming upon a semantics that omits tail-call optimization is unacceptable.

In the presence of tail-call optimizations, a central theme of this chapter is to recognize that *cflow* has a control effect. Therefore, we consider two implementations of dynamic join points. The first implementation corresponds to the original AspectScheme specification, where dynamic join points are given in context made available by continuation marks. In this first case, we identify the unsound optimizations mandated by the Scheme standard [91]. The second implementation relies on the aspect programmer to accumulate context. In this second case, we note that tail-call optimizations are sound, and recognize that a state effect is entailed.

3.2 AspectScheme with *Cflow*

The first step in applying advice is to generate dynamic join points. The definition of dynamic join points is given via the $J[\![\bullet]\!]$ function, shown in Figure 23. It's purpose is to examine the defunctionalized continuation and extract the parts of interest. In this section, the parts of interest are procedure applications which are in-progress, along with the procedure being applied and the argument values at that application site. In order to retain the information about the pending calls, we extend the continuation with an additional frame to store the procedure and the arguments it is being applied to. When the body of the procedure completes, the result value is applied to the continuation-mark frame, which simply applies the result value to the next frame in sequence. This is formally expressed in Figure 22.

This continuation-traversing implementation of J is given in Figure 23.

The second step in applying advice is to match the advice pointcut against the dynamic join point and weave as appropriate. We do this for each advice in the advice environment,

$$A = \{\langle scope, pc^i, adv^i \rangle \mid i = 1, \ldots, \mid A \mid\}.$$

This entails applying pc^i to the dynamic join point in context ($jp*$). If this returns `false`, we return the original (untransformed) procedure. Otherwise, the pointcut

$$\begin{aligned}
&\langle VC_{\text{arg}}, \langle \textsf{app2-k}\ \langle (\lambda\,(x)\ M)_t, E_{\text{fun}}, A_{\text{fun}} \rangle, E_{\text{app}}, A_{\text{app}}, K \rangle, S \rangle \\
&\qquad \blacktriangleright_{app}\ \langle \langle M', E', A_{\text{app}} \rangle, K', S' \rangle \\
&\qquad \text{where} \\
M' &= (\textbf{app/prim}\ W[\![\mid A_{\text{app}} \mid]\!]\ \mathit{arg}) \\
K' &= \langle \textsf{markapp-k}\ \langle (\lambda\,(x)\ M)_t, E_{\text{fun}}, A_{\text{fun}} \rangle, VC_{\text{arg}}, K \rangle \\
\langle E', S' \rangle &= \langle E_{\text{app}}, S \rangle \\
&+ \{\mathit{fun} \mapsto \langle (\lambda\,(x)\ M)_t, E_{\text{fun}}, A_{\text{fun}} \rangle,\ \mathit{arg} \mapsto VC_{\text{arg}},\ \mathit{jp*} \mapsto J[\![K']\!]\} \\
&+ \{\mathit{pc}^i \mapsto VC_{\text{pc}^i}, \mathit{adv}^i \mapsto VC_{\text{adv}^i} \mid \langle \mathit{scope}, VC_{\text{pc}^i}, VC_{\text{adv}^i} \rangle \in A_{\text{app}}\}
\end{aligned}$$

$$\langle VC, \langle \textsf{markapp-k}\ VC_{\text{fun}}, VC_{\text{arg}}, K \rangle, S \rangle\ \blacktriangleright_{mark}\ \langle VC, K, S \rangle$$

Figure 22: CEKS Continuation Marking Application Rules

$$\begin{aligned}
J[\![\textsf{mt-k}]\!] &= \langle \textsf{empty}, E_0, A_0 \rangle \\
J[\![\langle \textsf{markapp-k}\ VC_{\text{fun}}, VC_{\text{arg}}, K \rangle]\!] &= \langle (\textsf{cons}\ (\textsf{cons}\ VC_{\text{fun}},\ VC_{\text{arg}}),\ J[\![K]\!]), E_0, A_0 \rangle \\
J[\![\langle \ldots, K \rangle]\!] &= J[\![K]\!] \text{ otherwise}
\end{aligned}$$

Figure 23: CEKS Dynamic Join Point Construction 1

will have returned a list of context arguments. We apply the corresponding advice adv^i to the context arguments to yield a procedure transformer. That transformer is applied to the original function yield a new, advised function. The transformation, W, with base case of the original function, *fun*, is given in Figure 24.

Notice the following points about W:

1. If no advice exists, it simply returns the original function, which will be applied, using **app/prim**, to the argument.

2. It applies each pointcut to the dynamic join point.

$$
\begin{array}{lcl}
W[\![0]\!] & = & \mathit{fun} \\
W[\![i]\!] & = & (\textbf{app/prim}\ (\lambda\,(f)\ (\textbf{let}\ ([a\ (\textbf{app/prim}\ pc^i\ jp{*})])\\
& & \qquad\qquad\qquad (\textbf{if}\ a\\
& & \qquad\qquad\qquad\quad ((\textbf{app/prim}\ adv^i\ f)\ a)\\
& & \qquad\qquad\qquad\quad f)))\\
& & \qquad\quad W[\![i-1]\!]) \qquad\qquad\qquad \text{for } i > 0
\end{array}
$$

Figure 24: CEKS Advice Weaving

3. If no pointcut holds, again it returns the original function.
4. If some pointcuts hold, then it uses **app/prim** to apply the final transformed procedure to the original argument. Note that applications in the body of the transformed procedure may also invoke advice.

Third, we take the procedure resulting from all applicable advice transformations and **app/prim** it to the original argument, yielding a new expression for this function application:

$$M' = (\textbf{app/prim}\ W[\![\,|\ A_{\text{app}}\ |\,]\!]\ arg)$$

The third transition rule applies advice as described above, binding the various variables for the function, *fun*, the argument, *arg*, dynamic join point, $jp{*}$, and all advice components (pc^i and adv^i) in the environment and store. Evaluation moves to the new M', carrying the dynamic advice environment A_{app} for use within M', but the static advice environment remains available as part of the *fun* closure.

Cflow and Optimizations

With the entire pending call structure available through $J[\![\bullet]\!]$, implementing the *cflow* pointcut is straightforward, as seen in Figure 25. The J dynamic join point constructor has supplied a list of the in-progress calls in the continuation. We simply traverse the list, looking for matches.

```
(define ((cflow pc) jp*)
  (and (not (empty? jp*))
       (or (app/prim pc jp*)
           (app/prim (app/prim cflow pc) (rest jp*)))))
```

Figure 25: CEKS Cflow Pointcut Implementation

Our implementation has eliminated Scheme's tail-call optimizations. In the standard, Scheme implementations are required to be "properly tail-recursive", meaning that "an infinite number of active tail calls" must be supported [30; 91]. However, our implementation cannot do this because every active tail call accumulates additional continuation frames. Fortunately, we have no other option, because we must carry the additional continuation-mark frames; otherwise the behaviour of the program in Figure 26 is incorrect. We expect that program to print

```
(4 1) (3 4)
(3 4) (2 12)
(2 12) (1 24)
```

Without the context contained in the continuation-mark frames, each recursive call to *fact* is a tail call and the enclosing call is optimized off the continuation stack. Hence, with tail call optimization, we would get no printed output. Tail call optimization is unsound if dynamic join points include context.

Because we (intentionally) destroy tail-call optimization, our approach suffers from a run-time penalty. Given that **cflow** and **cflowbelow** pointcuts can discriminate the number and order of calls, it is straightforward to see that this cannot be improved to full tail-call optimization. In languages like AspectJ (sans **args** except for the top-most dynamic join point), the entire range of interesting continuation-mark sequences is known in advance. In that case, a regular automaton can recognize the dynamic join points [144], and the actual continuation marks need only denote the current automaton state. Thus, phased implementations like ajc [84], and abc [12], can restore tail-recursive optimizations for procedure calls which do not alter the automaton state; the process is similar to that described by Clemens and Felleisen [24; 25].

AspectScheme supports capturing arguments from *cflow* dynamic join points; therefore a regular automaton no longer suffices: a push-down automaton is

```
;; accumulator-style tail recursive factorial
(define (fact n a)
  (if (zero? n)
      a
      (fact (- n 1) (* n a))))
(let ([(((adv proceed) n1 a1 ) n a)
       (print (n1 a1) (list n a))
       (newline)
       (app/prim proceed n a)])
  (around (&& (&& (cflowbelow (call fact))
                  args)
              (&& (exec fact) args))
          adv
     (fact 4 1)))
```

Figure 26: Tail-recursive Factorial and Advice

required. Now, there clearly is context building up and tail-call optimization is manifestly impossible in the general case.

3.3 State Effects Cflow

In this section, we pare our dynamic join point model back to that of Chapter 2. That is, we eliminate context from a dynamic join point and show that tail-call optimization is restored. Then, we show how to recover *cflow* constructs using state effects. Again, we examine the three steps for applying advice:

1. generate the dynamic join point,
2. match it to the pointcut, and
3. apply the advice body as appropriate.

The first step in applying advice is to generate dynamic join points. In this implementation, the dynamic join point does not carry context. Therefore, the continuation K does not need to be extended with a continuation mark. Also, the $J[\![\bullet]\!]$ function is considerably simpler, needing only to examine the closure in the top continuation frame, $\langle \mathsf{app2\text{-}k}\ \langle (\lambda\,(x)\ M)_t, E_{\text{fun}}, A_{\text{fun}}\rangle, E_{\text{app}}, A_{\text{app}}, \rangle$and the applied value. This is illustrated in Figure 27.

$$\begin{array}{rcl}
\multicolumn{3}{l}{\langle VC_{\text{arg}}, \langle \textsf{app2-k}\ \langle (\lambda\,(x)\ M)_t, E_{\text{fun}}, A_{\text{fun}}\rangle, E_{\text{app}}, A_{\text{app}}, K\rangle, S\rangle} \\
& \blacktriangleright_{app} & \langle\langle M', E', A_{\text{app}}\rangle, K, S'\rangle \\
& \text{where} & \\
M' & = & (\textbf{app/prim}\ W[\![\mid A_{\text{app}} \mid]\!]\ arg) \\
\\
\langle E', S'\rangle & = & \langle E_{\text{app}}, S\rangle \\
& + & \{fun \mapsto \langle (\lambda\,(x)\ M)_t, E_{\text{fun}}, A_{\text{fun}}\rangle,\ arg \mapsto VC_{\text{arg}},\ jp \mapsto J[\![(\lambda\,(x)\ M)_t]\!]\} \\
& + & \{pc^i \mapsto VC_{\text{pc}^i}, adv^i \mapsto VC_{\text{adv}^i} \mid \langle scope, VC_{\text{pc}^i}, VC_{\text{adv}^i}\rangle \in A_{\text{app}}\}
\end{array}$$

Figure 27: CEKS Application Rules 2

$$J[\![f]\!] \quad = \quad f$$

Figure 28: CEKS Dynamic Join Point Construction 2

It is important to note that the continuation stack is strictly smaller in this implementation of application, yielding the needed tail-call behaviour.

The simpler implementation of J is given in Figure 28. It declares that a dynamic join point is only the current function to be applied. We no longer need to traverse each continuation frame to accumulate dynamic join point context.

The second step in applying advice is to match the advice pointcut against the dynamic join point and weave as appropriate. We do this for each advice in the advice environment,

$$A = \{\langle scope, pc^i, adv^i\rangle \mid i = 1, \ldots, \mid A \mid\}.$$

This entails applying pc^i to the dynamic join point in context (jp). If this returns false, we return the original (untransformed) procedure. Otherwise, the pointcut will have returned a pair comprising the function and the argument. We apply the corresponding advice adv^i to these values to yield a procedure transformer. That transformer is applied to the original function yield a new, advised function. The transformation, W, with base case of the original function, *fun*, is given in

$$
\begin{aligned}
W[\![0]\!] &= \mathit{fun} \\
W[\![i]\!] &= (\textbf{app/prim}\ (\lambda\,(f)\ (\textbf{if}\ (\textbf{app/prim}\ pc^i\ jp) \\
&\qquad\qquad\qquad\qquad (\textbf{app/prim}\ adv^i\ f) \\
&\qquad\qquad\qquad\qquad f)) \\
&\qquad\quad W[\![i-1]\!]) \qquad\qquad \text{for } i > 0
\end{aligned}
$$

Figure 29: CEKS Advice Weaving 2

Figure 29.

This weaver differs from the previous one only in that context arguments are no longer provided and applied as part of the transformation process. This is as expected, given that *cflow* is not available natively.

The third step is to take the procedure resulting from all applicable advice transformations and **app/prim** it to the original argument, yielding a new expression for this function application:

$$M' = (\textbf{app/prim}\ W[\![\mid A_{\text{app}} \mid]\!]\ \mathit{arg})$$

The third transition rule applies advice as described above, binding the various variables for the function, *fun*, the argument, *arg*, dynamic join point, *jp*, and all advice components (pc^i and adv^i) in the environment and store. Evaluation moves to the new M', carrying the dynamic advice environment A_{app} for use within M', but the static advice environment remains available as part of the *fun* closure.

Regenerating Cflow

Now we turn our attention to providing the equivalent functionality that *cflow* gave us originally. We offer two distinct techniques for doing this, each embodied as a translation of cflow into this simpler AspectScheme dialect. By examining these translations, we will identify the control effect that cflow engenders.

In this section, we consider a pointcut containing *cflow*. Without loss of generality, it is structured as (*&&* (*cflow* PC_C) PC), where PC may be empty. Our model advice is given in Figure 30.

```
(around (&& (cflow PC_C) PC)
            (λ (args_C)
              (λ (jp)
                (λ (args
                  body)))
   ...)
```

Figure 30: Cflow Model Advice

```
(around PC_C
            (λ (jp_C)
              (λ (args_C)
                (fluid-around PC
                                (λ (jp)
                                  (λ (args
                                    body))
                  (app/prim jp_C args_C))))
   ...)
```

Figure 31: Invalid Cflow Translation

Cflow by Dynamic Advice Introduction

One potential translation of *cflow* is given in Figure 31.

The idea behind this implementation is that a *cflow* pointcut specializes an advice so as to have different behaviour inside and outside a matching dynamic join point. Unfortunately, *cflow* only recognizes the closest enclosing dynamic join point. So, for a simple example:

```
(define (cf-fun x)
  (if (= x 0)
     (cf-fun (- x 1)))
  (fun))

(define (fun) (display 'fun))
```

```
(around (&& (&& (cflow (call cf-fun))
                (args))
            (call fun))
        (λ (x)
          (λ (jp)
            (λ (#;no-args)
              (display x)
              (app/prim jp))))
  (cf-fun 1))
```

the displayed result will be

```
0 1 fun 1 fun
```

rather than

```
0 fun 1 fun
```

What has happened is that two advice are applied in the translated version, whereas only one is applied in the original AspectScheme semantics. In order for this to work, some sort of search *cut* operation for nesting related advice is required. The following, correct, translation captures this cut as a state effect.

Cflow by Effects

Masuhara et al. [114] identified another translation of the stack-crawling implementation of *cflow*. In that paper, we used partial-evaluation to validate a model where each *cflow* pointcut is represented as a separate context stack. We show the AspectScheme variation of this in Figure 32.

Our model accounts for this optimization. It recognizes that the behaviour is to restore tail-call optimizations by incorporating a side-effected stack and the necessary operations around each dynamic dynamic join point matching the *cflow* pointcut argument.

In the original AspectScheme implementation, every procedure application needed to mark the continuation to ensure that the calling structure was available for *cflow* pointcuts to match against. In practice, however, only a subset of the procedure calls actually required this breaking of proper tail-recursion. In this translation, only those calls matching PC_C have this improper tail recursion. Advice has altered their behaviour to incorporate additional behaviour, (**set!** *stk* (*cdr stk*)), which mutates a cell in the store.

```
(let ([stk '()])
  ([adv (λ (jp)
          (λ (args)
            (if (null? stk)
                (app/prim jp args)
                (let ([args_C (car stk)])
                  body))))])
   (around PC_C
           (λ (jp_C)
             (λ (args_C)
               (set! stk (cons args_C stk))
               (app/prim jp_C args_C)
               (set! stk (cdr stk))))
     (around PC_2
             adv
       ...)))
```

Figure 32: Valid Cflow Translation

This translation technique also clarifies one difficult situation for *cflow* identified in our AspectScheme article [58].

```
(let ([f (λ (g)
           (let ([h (λ () 1)])
             (around (&& (call h)
                         (cflow (call g)))
                     (λ (jp)
                       (λ ()
                         (+ 1 (jp))))
                     (h))))])
  (f f))
```

The application of f to itself makes the *cflow* (*call* g) pointcut true. Therefore, the advice within f should be applied, requiring us to remember the calling context even in the absence of any advice at the time of the call to f. AspectScheme, as given in Section 3.2 maintains the entire calling context, and therefore does apply the advice. The translated version would need the **let** ([*stk* . . .]) to be hoisted outside the call (f f) making it explicit that it covers the entire expression.

3.4 Related Work

Two groups have explored the idea of aspects in higher-order languages, specifically the ML family [118].

One group has developed Aspectual CaML [115] which incorporates aspects for pointcuts and advice, and intertype declarations into the ML [118] fragment of Ocaml. They support similar dynamic join points to AspectScheme, as well as ones related to structure creation and matching. Although Ocaml is a higher-order language, pointcuts and advice are not higher-order in their variant. They work from an expression-based (syntax-derived) understanding of aspects, and implement the weaver as compile-time rewriting. This differs from our work where we recognize the semantic foundations of AOP, built on structured continuations. They also omit the thorny issue of *cflow*, only providing the lexical version, *within*. They do tackle one challenging problem, that of inferring and statically checking types for aspects, which our latently-typed system ignores.

The MiniAML system [165], discussed earlier, also provides a higher-order language with pointcuts and advice aspects. Their system translates a stripped-down functional fragment of ML into a lower-level system, $\mathbb{F}_A$. In this target language, labeled evaluation contexts correspond to our marked continuations. Pointcuts are higher-order label matchers; advice inserts additional behaviour. However, they only support before and after advice. Hence, cflow and more general aspects are not included. This simplifies their static analysis, which focuses on type-inference and checking. Surprisingly, they comment that polymorphic around advice appears to be intractable.

Both ML-based systems provide aspects for the expression-based core language, ignoring the modules and functors part of the standard. We also ignore modules for AspectScheme, because there is currently no standard.

Another system, SteamLoom [18], provides facilities similar to our higher-order pointcuts and advice. It is an extended Java virtual machine that supports dynamic aspect instantiation and execution. Our AspectScheme implementation provides a similar facility by allowing pointcuts and advice to be passed as arguments, and installed dynamically.

3.5 Summary

This concludes our dynamic semantics account of dynamic join points, pointcut, and advice. In a first-order procedural framework, we moved from an expression-based understanding of dynamic join points, pointcuts, and advice, to a continuation-based understanding of these constructs. In doing so, our model supports a principle for dynamic join points, elevating them from "well-defined" points in an execution to "principled" points. This has enabled us to directly expose well-founded dynamic join points, pointcuts, and advice within a standard language, Scheme, and to identify and resolve issues regarding aspect scoping and tail-call optimization.

Our investigation of optimizations has given us a glimpse of the next part the dissertation, where we try to understand the static semantics of dynamic join points, pointcuts, and advice AOP. In particular, our implementation of *cflow* identifies state as the fundamental effect that its advice must supply.

Part II

Static Semantics

Chapter 4

Abstracting Pointcuts and Advice to Effects

A value is, a computation does.

Paul Blain Levy
Call-By-Push-Value [104]

In the preceding part of this dissertation, we have presented a novel dynamic semantic construction for dynamic joinpoints, pointcuts, and advice. Our goal in this part is to examine the corresponding static semantics. This examination is divided across two chapters. This chapter describes an effect analysis [125][8] for the Proc language, especially the novelty of effect descriptions for pointcuts. The next chapter will apply these analyses to characterize the interaction of pointcuts and advice, and compare our results to similar work.

Static Semantics Static semantic descriptions are focussed on providing an abstraction of program constructs, which can be applied to one of three main goals:

optimize compilation — identify and validate compilation optimizations, such as code inlining, closure conversion, code motion, and dead-code elimination;

8. Also referred to as *behaviour* analysis.

optimize execution — identify and validate runtime optimizations, such as stack allocation instead of heap allocation of variables, and elimination of dynamic type tests (especially in latently typed languages);

validate intent — document, and validate programmer intent; including checking type annotations, and inferring types and computational effects.

In this work we are focused on the third goal, for which the dynamic semantics presented in Chapter 2 is well suited.

We will also restrict our attention to expressing computational effects rather than on type inference and checking. Type inference and checking of dynamic join points, pointcuts, and advice is certainly a fruitful endeavour; Walker et al. [165] and Dantas et al. [41, 42] have closely investigated this subject, and recently discovered sophisticated local and global type-inference algorithms that make inference work. Effect checking has not received the same attention in the research community.

Furthermore, our novel semantic description highlights the understanding of dynamic join points as activations of continuation frames — *constructs that have an effect as well as a type*. Just as reasoning about continuations led to effect analysis (c.f. Jouvelot and Gifford [86] and Sabry and Felleisen [141]), we believe pointcuts and advice naturally admit an effect-based description. In the next chapter, we examine the additional insight that develops from characterizing dynamic join points, pointcuts, and advice in terms of effects.

4.1 Computational Effects

Before we develop our effect analysis, it is instructive to briefly describe computational effects. Below, we enumerate a number of fundamental computational effects which various researchers have identified and studied. Based on work beginning with Moggi [120] and continuing to recent work [143; 163; 164], we adopt the monadic taxonomy of effects.

State — captured by monad $Ta = a * v$, the product type-constructor; where a is the original computation and v is a value containing the desired state.

Exception — captured by monad $Ta = a + e$, the disjoint sum type-constructor; where a is the original computation and e represents exceptions.

Concurrency — captured by monad $Ta = [a]$, the list type-constructor; providing multiple a computations interleaved by *bind* and *return*.

Nondeterminism — captured by the monad $Ta = [a]$, the list type-constructor; providing multiple a computations executed in-order by *bind* and *return*.

All of these effects correspond to a restricted subset of global program transformations. The restriction is that the transformation can be abstracted into an execution monad $\langle Ta, \text{return}, \text{bind} \rangle$ as outlined in Wadler [160]. The essential idea of a monad is that return provides an effect-free computation, and bind sequences two computations so that their effects are ordered properly.

Two or more effects can be combined in either simple or complex ways. For example, combining two state monads in the obvious way is isomorphic to providing a single state monad where v is a pair. Input/output can also be modeled as a combination of two state monads, but with a stronger coupling: all output computations that appear before an input computation must be sequenced to complete before the input computation. This provides the expected interactive behaviour.

Different effects can also be combined in different ways; and the order of combination can yield dramatic differences. For example, combining exceptions and state can yield two different behaviours. If exceptions wrap a stateful computation (i.e. $Ta = (a * s) + e$), then an exception will drop state changes, resulting in transaction-like behaviour. Most imperative programming languages wrap state around exceptions (i.e. $Ta = (a+e)*s$), retaining state changes in the event of an exception. Exceptions and concurrency also combine in two different layerings. If concurrency wraps exceptions, then exceptions are isolated to each thread and do not impact the other threads. Alternately, exceptions wrapping concurrency supports an exception aborting all threads.

Our Proc language provides an effect model summarized as

$$Ta = (([(a + e)] * s_{\text{global}}) + e_{\text{abort}}) * \mathit{output}$$

State is global only. Exceptions do not reset global state nor output; and **abort** discards all execution but leaves output intact.

This is essentially the effect model of Java[9], summarized as

$$Ta = ((([(a + e)] * s_{\text{global}}) + e_{\text{abort}}) * output$$

Exceptions are isolated to individual threads. Global state is shared across all threads, but is not preserved if the entire system is aborted. Last, output is preserved even if the program is aborted.

The goal of our effect analysis is to supply a report of the effects of procedures, pointcuts, and advice. From these reports, programmers may identify unexpected pointcut properties and unusual interactions between program code and advice.

4.2 Effect Analysis for Proc

First, we provide a basic behaviour analysis for Proc, predicated on our continuation-based dynamic semantics. That system captures input/output, state, and sequencing provided respectively the *read* and *display* primitives, the *get* and *set* expressions, and the *seq* expression. Later, we extend it with additional effects, namely exceptions and threads, in order to provide a more robust set of effect descriptions.

As we are working with continuations, we use a continuation-passing style intermediate language. This construction originated with with Appel [10], and was applied by Shivers [147] as the basis for control- and data-flow analysis in Scheme. Contemporaneously, Harrison [81] inaugurated *procedure string* to annotate the abstract effects of procedures. Our effect string descriptions will be a variation of these procedure strings.

Effect Strings for Procedures

We begin by computing effect descriptions of procedures. We provide a straightforward implementation of behaviour analysis. Continuations are abstracted into *pureS*, representing pure computations, *bindS*, representing sequencing of two potentially effect-ful expressions, and a variety of expression-specific abstract continuations. This unfolding operation relies on a standard codewalker, shown in Figure 33.

9. Note that s_{local} is provided indirectly; each thread-local variable exists as a `Map` data structure, with values keyed by thread objects.

```
(define (walk f x)
  (f (cond [(litX? x) x]
           [(varX? x) x]
           [(ifX? x) (make-ifX (walk f (ifX-test x))
                               (walk f (ifX-then x))
                               (walk f (ifX-else x)))]
           [(seqX? x) (make-seqX (map (λ (x) (walk f x))
                                      (seqX-exps x)))]
           [(letX? x) (make-letX (letX-ids x)
                                 (map (λ (x) (walk f x))
                                      (letX-rands X))
                                 (walk f (letX-body x)))]
           [(getX? x) x]
           [(setX? x) (make-setX (setX-id x)
                                 (walk f (setX-rand x)))]
           [(appX? x) (make-appX (appX-id x)
                                 (map (λ (x) (walk f x))
                                      (appX-rands x)))]
           [(pcdX? x) (make-pcdX (map (λ (x) (walk f x))
                                      (pcdX-rands x)))]
           [else (error 'parse "not an exp: ~a" s)])))
```

Figure 33: Expression Codewalker

```
;;; shadow frames – abstraction of continuations
;; auxiliary shadow frames
(define-struct pureS [val])
(define-struct bindS [1st 2nd])
(define-struct bind2S [1st 2nd 3rd])

;; join point shadow frames
(define-struct getS [id])
(define-struct setS [id])
(define-struct callS [id])  ; includes exec

;; for advice analysis
(define-struct pcdS [rands]))
```

Figure 34: Shadow Frames

```
(define (unfold x)
  (walk (λ (x)
          (cond [(litX? x) (make-pureS)]
                [(varX? x) (make-pureS)]
                [(ifX? x) (make-bindS (ifX-test x)
                                      (make-bind2S (ifX-then x)
                                                   (ifX-else x)))]
                [(seqX? x) (let loop ([exps (seqX-exps x)])
                             (cond [(null? exps) (make-pureS)]
                                   [(null? (cdr exps)) (car exps)]
                                   [else (make-bindS (car exps)
                                                     (loop (cdr exps)))]))]
                [(letX? x) (foldr make-bindS
                                  (letX-body x)
                                  (letX-rands x))]
                [(getX? x) (make-getS (getX-id x))]
                [(setX? x) (make-bindS (setX-rand x))
                                       (make-setS (setX-id x)))]
                [(appX? x) (foldr make-bindS
                                  (make-appC (appX-id x))
                                  (appX-rands x))]
                [(pcdX? x) (foldr make-bindS
                                  (make-pcdC (pcdX-rands x))
                                  (pcdX-rands x))]
                [else (error 'parse "not an exp: ~a" s)]))
        x)
```

Figure 35: Unfolding Expressions into CPS Form

The *unfold* operation terminates, yielding a structure abstracting the sequence of effect-ful operations comprising the expression. For procedure declarations, we abstract their procedure body expressions and compress this sequence into a *procedure string* [81]. Compression is done by walking the CPS intermediate language and accumulating the effects present. This yields a tuple

$$\langle I, O, G, S \rangle$$

describing

I — input effect (*read* primitives),

O — output effect (*display* and *newline* primitives),

G — state access (*getS* abstract frames),

S — state mutation (*setS* abstract frames).

In the case of the latter two items, sets of globals that are read and mutated are kept. These sets are simple instances of *regions*; attributes of effects that detail limitations of effects. In this case, the region indicates the subset of the entire store. We will denote the effect string for a pure computation as

$$E_0 = \langle \#\mathsf{f}, \#\mathsf{f}, \emptyset, \emptyset \rangle$$

Last, we combine the effects of procedure calls. We walk the CPS intermediate form for each procedure declaration repeatedly, constructing more complete values for E_D by taking the union of the original procedure's tuple and each called procedure's tuple. This iterative process will terminate, because the effect combination is idempotent.

Theorem. *The translation to* CPS *intermediate form and the calculation of effect strings terminates.*

Proof Sketch: Translation strictly monotonically reduces the number of subexpressions remaining. The process of iteratively combining effect strings strictly monotonically increases the number of procedure declarations subsumed into each procedure string. Since these are finite, termination is assured.

Effect Strings for Dynamic Join Points

We have kept all of this simple, because our attention is not on developing new effect analyses, but on dynamic join points, pointcuts, and advice. Therefore, within this framework, we extend the effect analysis to the three aspect-oriented constructs in our language, beginning with dynamic join points, shown in Table 1.

Dynamic join points are not actually present in our CPS intermediate language; only their *shadows* are. Dynamic join point shadows are the static abstraction of dynamic join points. These shadows appear as our abstract (shadow) frames in the CPS intermediate language. Auxiliary frames correspond to *bindS*

$$
\begin{aligned}
E_D(i) &= E_S[\![body]\!] \\
& \quad \text{where} \\
& \qquad (i\ (proc\ ids\ body)) \text{ is a procedure declaration} \\[1ex]
E_S[\![(pureS\ v\ldots)]\!] &= E_0 \\
E_S[\![(bindS\ x_1\ x_2)]\!] &= E_S[\![x_1]\!] + E_S[\![x_2]\!] \\
& \quad \text{where} \\
& \qquad \langle I_1, O_1, G_1, S_1 \rangle + \langle I_2, O_2, G_2, S_2 \rangle \\
& \qquad\quad \equiv\ \langle I_1 \mid I_2, O_1 \mid O_2, G_1 \cup G_2, S_1 \cup G_2 \rangle \\[1ex]
E_S[\![(bind2S\ x_1\ x_2\ x_3)]\!] &= E_S[\![x_1]\!] + E_S[\![x_2]\!] + E_S[\![x_3]\!] \\[1ex]
E_S[\![(getS\ i)]\!] &= \langle \#\mathsf{f}, \#\mathsf{f}, i, \emptyset \rangle \\
E_S[\![(setS\ i)]\!] &= \langle \#\mathsf{f}, \#\mathsf{f}, \emptyset, i \rangle \\
E_S[\![(callS\ i)]\!] &= E_P(i) \\
E_S[\![(execS\ i)]\!] &= E_P(i) \\[1ex]
E_P(\texttt{read}_{\text{prim}}) &= \langle \#\mathsf{t}, \#\mathsf{f}, \emptyset, \emptyset \rangle \\
E_P(\texttt{display}_{\text{prim}}) &= \langle \#\mathsf{f}, \#\mathsf{t}, \emptyset, \emptyset \rangle \\
E_P(\texttt{newline}_{\text{prim}}) &= \langle \#\mathsf{f}, \#\mathsf{t}, \emptyset, \emptyset \rangle \\
E_P(i_{\text{prim}}) &= E_0 \text{ for other primitives} \\
E_P(i_{\text{proc}}) &= E_D(i) \text{ for user-declared procedures}
\end{aligned}
$$

Figure 36: CPS Intermediate Language Effects

Dynamic Join
$(loc\ i_g) \blacktriangleright (getF)$
$(loc\ i_g) \blacktriangleright (setF\ val)$
$(val\ \ldots) \blacktriangleright (callF\ i_p)$
$(proc\ i_p) \blacktriangleright (execS\ val\ldots)$

Table 1: Dynamic Join Points

and *bind2S* abstract frames. The other frames that are activated at a dynamic join point are represented individually:

- *getS* – for *getF* dynamic join points
- *setS* – for *setF* dynamic join points
- *callS* – for *callF* and *execF* dynamic join points.

The correspondence is shown in Table 2.

In rare circumstances, a shadow may correspond to one dynamic join point; but, they usually abstract many dynamic join points. To see this, consider a recursive procedure call. The recursive call expression will be CPS-translated into a single *callS* shadow. At execution, each recursive call will generate a new *callF* continuation frame; activating each of these frames is a new dynamic join point.

Dynamic join points in Proc are *get* and *set* of global variables, or *call* and *exec* of procedures; reified as applications of values to the appropriate continuation frames. Each of these dynamic join points has a well-known effect given either as the state operation and associated region (in the case of *getF* and *setF*) or the effect computed for the procedure (in the case of *callF* and *execF*). Therefore, we can attach these effects to the shadows as well[10]. We will use these to determine an effect description of pointcuts.

Effect Strings for Pointcuts

Although pointcuts do not have effects, they identify dynamic join points which do have effects. In the case of pointcuts, we do not have a single procedure string representing the pointcut effect. Instead, because pointcuts may correspond to multiple dynamic join point shadows, we maintain a list of the effects corresponding to the shadows. Also, for use later in advice bodies, we also maintain a list

Dynamic Join Point	**Shadow**
$(loc\ i_g) \blacktriangleright (getF)$	$(getS\ i_g)$
$(loc\ i_g) \blacktriangleright (setF\ val)$	$(setS\ i_g)$
$(val\ \ldots) \blacktriangleright (callF\ i_p)$	$(callS\ i_p)$
$(proc\ i_p) \blacktriangleright (execS\ val\ldots)$	$(callS\ i_p)$

Table 2: Dynamic Join Point Shadows

10. Note that the effects associated with arguments to procedure calls and global sets are not included in the effect of the dynamic join point. This matches the behaviour of the CPS semantics, where evaluation of arguments is not part of the dynamic join point.

$$
\begin{aligned}
E_C[\![(getC\ i_g)]\!] &= (E_S[\![(getS\ i_g)]\!], \emptyset) \\
E_C[\![(setC\ i_g\ i_v)]\!] &= (E_S[\![(setS\ i_g)]\!], \{i_v\}) \\
E_C[\![(callC\ i_p\ i\ldots)]\!] &= (E_S[\![(callS\ i_p)]\!], \{i\ldots\}) \\
E_C[\![(execC\ i_p\ i\ldots)]\!] &= (E_S[\![(callS\ i_p)]\!], \{i\ldots\})
\end{aligned}
$$

$$
\begin{aligned}
E_C[\![(notC\ pc)]\!] &= (E_0, \emptyset) \\
E_C[\![(orC\ pc_1\ pc_2)]\!] &= ((E_1 \cup E_2), A_1) \\
& \quad \text{where} \\
& \qquad \langle (E_1, A_1) \rangle = E_C[\![pc_1]\!] \\
& \qquad \langle (E_2, A_2) \rangle = E_C[\![pc_2]\!]
\end{aligned}
$$

Figure 37: Inferred Pointcut Effects

of argument names from the pointcut. Therefore, the effect string for a pointcut is an ordered pair (E, V) where

E — is a list of the original four-tuples $\langle I, O, G, S \rangle$, one for each matched pointcut,

V — is the list of variable names to be bound from the dynamic join point[11]

Therefore, by examining dynamic join point shadows and the pointcut, we can associate effect descriptions with pointcuts, as displayed in Figure 37.

Pointcut Effect Reports

The first benefit of this effect analysis is now at hand. Our analysis allows us to report an effect description of pointcuts to the programmer – essentially a summarization of the effects expected at all the dynamic join points that match the pointcut.

One would expect that often these dynamic join point effects would be consistent. Since pointcuts match multiple dynamic join points, and an advice applies at all of those dynamic join points, it seems reasonable that the dynamic

11. We assume well-formed programs, hence the variable names are consistent across subpointcuts.

join point effects would be consistent. It appears unusual to have dramatically different behaviour among the dynamic join points identified by the pointcut. Furthermore, the effects of those dynamic join points would be expected to have similar effect annotations. For example, an advice designed to maintain consistency in a model-view-controller design would have pointcuts identifying changes to state variables in the model, and advice informing the view to update. The altered behaviour manifested by the advice must provide some new effect that suits all of the identified dynamic join points.

For each pointcut, we summarize the following global properties:

- whether all, some, or none of the sub-pointcuts' effect strings contain the input effect,
- whether all, some, or none of the sub-pointcuts' effect strings contain the output effect,
- the common region for state access effects — the global variables which are accessed in all sub-pointcuts,
- the common region for state mutation effects — the global variables which are mutated in all sub-pointcuts.

For each pointcut, we report

- the region of state access effects,
- the region of state mutation effects;

excluding the common region in each case. One avenue of research which we have only begun to pursue is to compare across sub-pointcuts; leading to abstractions such as "each sub-pointcut accesses and mutates a region disjoint from the other sub-pointcuts." This would seem more useful in a language with more structure to the global variables; for example, one designed to force programs to obey a Law of Demeter [107; 108]. This is because our language has no concept of limited knowledge about other units; the address space is a single, flat, global store. With more structure to the global store, e.g. partitioning into individual objects, then well-founded disjoint regions would appear. Then we could apply the Law of Demeter to these.

The report supplies the effect description for the pointcut and the effect descriptions for the various dynamic join point shadows. This contains enough information to highlight two unexpected situations:

1. Inconsistent effects at different dynamic join point shadows: that is the dynamic join points separate into sets with substantially different effects. This may be a situation where the programmer has advised superfluous dynamic join points.

2. Incomplete dynamic join point shadow sets: other dynamic join point shadows in the program may have the substantially the same effect, but are not identified by the pointcut. This may be a situation where the programmer has incorrectly omitted dynamic join points that ought to be advised.

It is important to stress that this analysis is unique for providing effect descriptions for pointcuts alone. No previous research offers this abstraction of pointcuts. We will examine this in more detail when we apply the effect analysis and compare/contrast with others' work in the next chapter.

Effect Strings for Advice Bodies

We now turn our attention to determining the effect behaviour of advice bodies. An advice body is superficially similar to a procedure body; but, it contains an additional kind of abstract frame, *pcdS* for representing *pcdF* applications generated by **proceed** expressions. Therefore, in addition to composing effects at the dynamic join point, advice bodies permit the same dynamic join point to proceed

- zero times,
- once,
- or more than once;

and these proceed points can be conditional or unconditional.

Our procedure string for advice bodies must recognize these unique capabilities. Hence, we provide two additional elements in the procedure string representing the effect of the advice body:

1. N_u — the number of unconditional **proceed** calls in the advice body, and

2. N_c — the number of contingent **proceed** calls in the advice body.

These are simple counting operations implemented for the codewalker, shown in Figure 38. Conditional proceeds occur within the consequents of an *ifX* expression; unconditional ones occur do not. If an *ifX* expression contains unconditional proceeds in both consequents, the minimum count will be carried forward as unconditional proceeds for the entire expression.

Our effect string for advice bodies is given by:

$$\begin{aligned} E_B[\![body]\!] &= \langle I, O, G, S, N_u, N_c \rangle \\ &\text{where} \\ &\quad \langle I, O, G, S \rangle = E_S[\![body]\!] \end{aligned}$$

The discrimination among number of proceeds is informative of the behaviour of the advice.

- Only conditional proceeds suggests that the effects of the dynamic join point are being masked by the advice body. Other portions of the program code that rely on the effects caused by the dynamic join point may become inoperative.

- A single unconditional proceed suggests that the advice body exists solely to compose additional effects at the dynamic join point. This causes the advice effects and the dynamic join point effects to be fused into a larger composed effect; these probably correspond to `before` or `after` advice in AspectJ.

- Multiple proceeds indicates that dynamic join point effects are occurring multiple times; this is significant for dynamic join points with output or state mutation effects.

The next chapter will utilize these reported values to characterize classes of pointcut and advice interactions.

Advice Effect Reports

Now that we have effect strings for pointcuts and for advice bodies, we can inform the programmer about the interaction behaviour between the advice and the control flow points identified by the pointcut.

```
(define (sum-proceeds c1 c2)
  (let-values ([(c1u c1c) c1]
               [(c2u c2c) c2])
              (values (+ c1u c2u)
                      (+ c1c c2c))))
(define (count-proceeds-helper x)
  (cond [(litX? x) (values 0 0)]
        [(varX? x) (values 0 0)]
        [(ifX? x) (let-values ([(tu tc) (ifX-then x)]
                               [(eu ec) (ifX-else x)])
                              (sum-proceeds (ifX-test x)
                                            (let ([n (min tu eu)])
                                              (values n
                                                      (max (+ (- tu n) tc)
                                                           (+ (- eu n) ec))))))]
        [(seqX? x) (foldr sum-proceeds
                          (values 0 0)
                          (seqX-exps x))]
        [(letX? x) (foldr sum-proceeds
                          (letX-body x)
                          (letX-rands x))]
        [(getX? x) (values 0 0)]
        [(setX? x) (setX-rand x)]
        [(appX? x) (foldr sum-proceeds
                          (values 0 0)
                          (appX-rands x))]
        [(pcdX? x) (foldr sum-proceeds
                          (values 1 0)
                          (pcdX-rands x))]
        [else (error 'count-proceeds "not an exp: ~a" x)]))
(define (count-proceeds x)
  (walk count-proceeds-helper x))
```

Figure 38: Proceed Counting

Our effect string for advice combines the pointcut and the advice body effect strings:

$$E_A[\![(advice\ pc\ body)\ =\ \langle I_C, O_C, G_C, S_C, V_C, I_B, O_B, G_B, S_B, C\rangle$$

where

$$\begin{aligned} E_C[\![pc]\!] &= (\langle I_C, O_C, G_C, S_C\rangle, V_C) \\ E_B[\![body]\!] &= \langle I_B, O_B, G_B, S_B, N_c, N_u\rangle \\ C &= \langle N_c, N_u, P\rangle \end{aligned}$$

and

$(i\ (advice\ pc\ body))$ is an advice declaration

where P indicates whether all **proceed**s will be called with the original values from the dynamic join point. This latter item utilizes information from the pointcut and the advice body. From the pointcut, we generate the list of identifiers just as in *match-pc*. In the advice body we check that the operands to each *pcdS* shadow are the same variables bound by the pointcut.

This data-flow analysis is simple and conservative; we could sharpen it to recognize cases such as:

- identical operand values returned by both branches of **if**,
- operand values that are stored into a global and then accessed from that global variable without mutation,[12]
- operand values that are rebound in a **let** and returned from that rebound variable,
- operand values that are returned as the last expression of a *seq*,
- application of procedures that would return the correct operand value unchanged,

and other typical situations. For our purposes, a more sophisticated data-flow analysis serves only to obscure the fundamental property that we want to highlight: whether the dynamic join point would proceed with the same arguments.

With this effect string, we answer the following questions:

12. This case would be unsafe if concurrency is permitted.

1. How many times is the dynamic join point proceeded conditionally?
2. How many times is the dynamic join point proceeded unconditionally?
3. Are the arguments to any proceed changed from the original values bound by the pointcut at the dynamic join point?
4. What is the overlap between the pointcut-common state-effect regions and the advice body state-effect regions?
5. What advice-body state-effect regions are disjoint from the corresponding pointcut state-effect regions?

The first two questions are answered directly from the advice body analysis. The remaining items result from combining properties of the pointcut and the advice body. The third question is answered by the previously noted data-flow analysis. The last two questions are answered by comparing the access and mutation regions for the pointcut and the advice body. If these are disjoint, then the advice interacts with the program in simple ways, as we will see in the next chapter. Alternately, if these regions have substantial overlap, then advice and dynamic join point effects are tangled together and may be difficult to reason about.

4.3 Exceptions and Threads

We extended the Proc language with exceptions to provide additional effects for analysis. This implementation is straightforward in a CPS framework, and follows the standard literature [14]; see Figure 39. Exceptions are denoted by identifiers;

(**define-struct** *tryX* [*body id hdlr*]) ; — TRY exp id exp
(**define-struct** *raiX* [*id*]) ; — RAISE id
(**define-struct** *frkX* [*thr*]) ; — FORK exp

Figure 39: Expressions for Exceptions and Threads

fork supplies a new empty continuation to the sub-expression. Concurrency is supplied by a round-robin scheduler and related code shown in Appendix C.

For our behaviour analysis, we need to extend the effect string to capture these additional effects. For exceptions, we add the following fields to the effect string:

X — the set of exceptions that might escape from this construct

Y — the set of exceptions that cannot escape from this construct

Clearly the existing frame shadows leave this new field unchanged[13]; but the new *try* expression and *raise* expression add and remove the exception identifier from X, respectively. In addition, we keep track of the covered exceptions, Y to identify which exceptions may be masked. To support thread-based concurrency, we twin the effect string, so that it now contains duplicate fields for effects that appear in another thread (i.e. arise from sub-expression to *fork*).

Our effect reports now encompass this additional information.

- Pointcut reports now identify whether all, some, or none of the dynamic join points might throw some common or individual set of exceptions.
- Advice reports now identify exceptions that can no longer escape; providing information about *effect masking* [86]. Similarly with state effects, we can report effect isolation, where state is related only to a particular advice; reducing the cognitive load for programmers.
- Advice also reports effects that occur in the main and the secondary thread.

Proceed interacts in interesting ways with *fork*. For example, in the following program the exception effect is pushed into a different thread which crashes with

```
(((thrower proc ()
          (throw checked))
  (adv advise (call thrower ())
       (fork (proceed ())))))
  (call thrower))
```

an un-handled exception.

13. Compressing the *bind2S* shadow simply merges the sets; our analysis does the same for both sub-effect strings.

When we were first exploring the feasibility of effect analyses for advice languages, we hand-examined a number of test cases in AspectJ. We tried the corresponding test case, where the advice does not annotate the checked exception in AspectJ. Surprisingly, it was not refused by the static type-checker, despite Java's static typing rules enforcing that all checked exceptions must be annotated. We discovered that AJDT 1.1.10 compiles to the Java virtual machine specification, which does not enforce exception annotations. Therefore, this program was not rejected by Java's static type-checking. Our principled semantic exploration of effects in dynamic join points, pointcuts, and advice languages has helped discover a subtle bug in a mainstream aspect language.

The results of our effect analysis allow us to highlight advice with unexpected behavioural interactions, including the following:

1. advice which push an exception into another thread, and out of the control flow of an exception handler,

2. advice which introduce new un-handled exceptions into a control flow,

3. advice which push state mutation into a new thread, allowing control flow to return before the expected update completes.

4.4 Effect Analysis for AspectScheme

Although this chapter uses Proc as the model for applying behaviour analysis, our extended language, AspectScheme is also amenable to this kind of analysis. Here we discuss which parts are tractable.

The problem of determining effects in Scheme is much more difficult than in our Proc language. In particular, control flow analysis is difficult to formulate when procedures are first-class. Shivers [147, 148] shows how to braid data flow and control flow analyses together to provide a tractable and useful effect analysis for Scheme. That same analysis can be extended to AspectScheme by recognizing advice as higher-order procedures.

Dynamic join points and pointcuts are more difficult to analyse in AspectScheme. Instead of simply providing a declarative language for identifying continuation structure, pointcuts in AspectScheme are higher-order user-defined procedures. They can inspect continuation marks as well as examine and mutate other program state. Hence the set of dynamic join points which a pointcut might iden-

tify is not computable in general. If we restrict ourselves to applications of the system-provided, pure pointcuts, *call*, *exec*, and *adv-exec*, then dynamic join point shadows are identifiable for AspectScheme. In this case, pointcuts support the same analysis that our model language, Proc, does.

Other imperative languages, such as C, Java, C++, etc. also include aliasing effects because values can be passed by reference (e.g. objects). Various existing data flow analyses [17; 125; 142] have been developed for these situations; and more research is continuing. Applying these is out-of-scope for this dissertation: it distracts us away from the static analysis of pointcuts and advice.

4.5 Related Work

Our effect analysis system draws heavily on three main sources: previous work on CPS-based program representations, previous work with effect analyses based on procedure strings, and the general body of types-and-effects analysis.

Our work is distinguished by our exploitation of CPS. Our static semantics is a direct abstraction of our dynamic semantics – the CPS frames translate to shadow frames, and dynamic join points lead to dynamic join point shadows. This gives a firm foundation to our analysis, and serves to develop our understanding of dynamic join point shadows.

Brooks et al. [19]; Steele [153] initiated the development of CPS-based compilation techniques, and introduced the first CPS intermediate language. Contemporaneously, Wand and Friedman [166] also investigated the utility of CPS for compilation. Shivers [147] used this facility to show that control flow analysis in the CPS frame work was tractable and useful. From there, many additional papers applying this framework have been written [77; 133; 134]. Flanagan et al. [73] has reduced the CPS intermediate language to A-normal forms and shown that many of the original analyses carry over. Most recently, Might and Shivers have re-examined the CPS intermediate forms to solve the *environment problem* and provide more powerful inter-procedural optimizations. Sabry [140] offers A-normal forms which, as noted in Chapter 2, eliminate some of the dynamic join points that aspect-oriented programming wishes to identify. Doing behaviour classification with a CPS intermediate language is novel; but it allows us to directly extend the analysis to support new control flow capabilities, such as threads and exceptions, that pointcuts and advice can manipulate.

Procedure string analysis was developed by Harrison [81]. His goal was to analyse Scheme programs, identifying inter-procedural optimizations and automate code parallelization. Many of the interactions that reduce opportunities for concurrency are the ones which our reports endeavour to highlight. Might and Shivers [117] extend these procedure strings to frame strings, as they move into a CPS intermediate language. Loucassen [110] and related papers [86; 111; 116] explore the effect behaviour supported by continuations, and consider various optimizations facilitated by this analysis. In particular, they developed the idea of effect masking, one of the obvious applications of dynamic join points, pointcuts, and advice.

Nielson et al. [125] are the primary reference for types and effects analysis. They provide a variety of data-flow, control-flow, and effect analyses over a simple procedural language similar to our Proc. The idea of regions for state effects is lifted directly from their model state analysis. These techniques are beginning to be applied to object-oriented languages; Skalka et al. [151] has recently provided a type-and-effect analysis for featherweight Java. A full Java language system is still unavailable. Lam et al. [99, 100] combine a number of pluggable analyses to provide control- and data-flow analyses, which they have applied to Java code.

Effects, as the *awkward squad* [129], are explicit in languages such as Haskell [83]. Therefore, they have encountered a wide variety of effects as monadic computations [105; 131; 162]. The various combinations have been explored in this monadic framework by a number of researchers: Espinosa [60], Steele [154], King and Wadler [95], and Jones and Duponcheel [85]. But they all relate back to Moggi's continuation-backed monadic models [119; 120], as described in a nice series of papers [46; 66; 67; 68] and by Wadler [161]. We lifted our range of computational effects from these sources.

Overall, our CPS intermediate language, and the abstraction to effects is a instantiation of Cousots' non-standard abstract interpretation framework of program analysis [32; 33; 34; 35; 36].

4.6 Summary

In this chapter, we have described our effect analysis implementation for Proc, supporting state and input/output effects, and an extended version that supports additional effects—exceptions and concurrency. By abstracting the CPS dynamic

semantics to a static CPS-based intermediate language we provide a natural development of dynamic join point shadows from the translation of continuation frames to shadow frames.

Our static semantics focuses on the novel property of continuations: that they carry computational effects. We

- characterize dynamic join point shadows by their input, output, state access and state mutation regions;
- associate dynamic join point effects with pointcuts, yielding reports summarizing and contrasting these effects;
- characterize advice bodies, reporting their input, output and state effects as well as repeated proceeding of dynamic join points
- compare and contrast the effects of pointcuts and advice by reporting the overlap between regions for each part.

Our effect analysis is conservative, losing precision compared to a collecting (execution) semantics. These include

- we compress effect strings for entire procedures, losing details about effect sequencing;
- we combine effect strings for both branches of a ifX, losing precision about path-dependent effects;
- we over-estimate the number of proceeds in an advice body by assuming the maximum;
- we partition concurrent effects into only two categories: this thread and all other threads.

Some restrictions, such as compressing effect strings, are necessary to ensure termination of our analysis. Others are mandated by working in a static framework where values which determine program flow are unavailable.

Our analysis is also for a simple language. An implementation for a mainstream language should include:

- input/output regions — represented by sets of channels or file handles; our language only has one input and output channel;

- data-flow analysis clarifying variable aliases — objects are a prime example of call-by-reference behaviour.

There is potential for robust research into this area. Supporting these additional analyses does not provide fundamentally new capabilities for effect analysis of aspect-oriented constructs; just more precision.

In the next chapter, we look closely at the reports generated by our effect analysis, and compare them with other AOP analyses.

Chapter 5

Classifying Pointcut and Advice Interactions

Pointcuts and advice admit a wide range of composition strategies, ranging from expected and acceptable to surprising and undesirable. For example, adding tracing operations to procedure calls, an output effect, seems clearly acceptable. Within a transactional context, this output behaviour would invalidate the all-or-nothing property expected of transactions: a clearly unacceptable result. The effect reports of the previous chapter highlight effect properties of the program, informing the developer of the interaction behaviour of their program.

These interactions appear in two situations:

- simple interactions, where advice interacts with the program (*advice* ***on*** *program*), and
- compound interactions, where more than one piece of advice applies at a dynamic join point (*advice* ***at*** *advice*).

In this chapter, we provide a classification system of simple interactions based on the components of the effect string. The classification proceeds along five axes:

1. control interactions — five categories,
2. data interactions — six categories,

3. input/output interactions — four categories,

4. exception interactions — three categories, and

5. concurrency interactions — three categories.

We show that our classification is a refinement of another accepted classification system given by Rinard et al. [138] showing the value of our effect analysis.

As shown in Rinard et al., effect characterization of aspects permits a classification system for aspect-oriented programs written in AspectJ, a mainstream AOP language. In this chapter, we offer an extended classification system, built upon the behaviour analysis developed in Chapter 4, that

- **characterizes** patterns of advice interaction,
- **automates** recognition of these patterns, and
- **highlights** for focused attention, those interactions with potentially surprising behaviour.

The basis of our classification system consists of a number of pairwise effect combinations which offer specific interactions. Our work is a refinement of the categorization given by Rinard et al. to incorporate concurrency and exceptions. We consider control flow interactions, data interactions, exception interactions, and concurrency interactions.

5.1 Simple Interactions

Simple interactions occur when an advice defines a new behaviour at a dynamic join point. The interaction is between the advice body and the dynamic join point identified by the pointcut. The dynamic join point may arise within the main program expression, a procedure body, or an advice body.

Control Flow Categories

The classification system for direct interaction focuses on control flow elements that affect how and when the advised dynamic join point executes. We identify five distinct kinds of interaction based upon the proceed behaviour of the advice body. These are summarized in Table 3, where C is given as in Equation 1 on Page 67.

Coupling The advice body permits the dynamic join point to proceed, unconditionally, exactly once, with the original arguments. This style of interaction, where the advice body *couples* the original behaviour of the dynamic join point, matches the application of **before** and **after** advice in languages such as AspectJ. Instrumentation aspects, such as tracing, monitoring, and subject-observer enforcement fit into this category.

Extension The advice body permits the dynamic join point to proceed, unconditionally, exactly once, with (potentially) different arguments. This style of interaction, where the advice body *extends* the behaviour of the dynamic join point, matches the application of **before** and **after** advice in languages such as PolyAML. Safety aspects which offer defaults, such as move-limiting and clipping fit into this category.

These two categories were originally captured by the single class, *augmentation*, by Rinard et al.[14] Our analysis more finely divides this category, based on whether arguments are passed unchanged through to the proceed expression. Both styles are present in existing AOP languages.

Narrowing The advice body permits the dynamic join point to proceed, conditionally, exactly once. Because the dynamic join point behaviour is not always maintained, the advice body *narrows* the behaviour of the dynamic join point. Safety aspects which ensure security or consistency before allowing the dynamic join point to proceed fit into this category.

Replacement The advice body never permits the dynamic join point to proceed. Instead, the behaviour of the advice body *replaces* the behaviour of the dynamic join point. This category contains aspects to disable portions of a system without removing the program code.

Repetition This catch-all category gathers advice bodies which *repeat* the dynamic join point more than once in any combination. Rinard et al. termed this *combination*, but our designation seems more informative. Regardless, this control flow interaction is one which merits programmer attention, as the effects

14. Rinard et al. also requires that exceptions cannot be thrown by the advice; we examine that portion of the analysis later in this chapter.

corresponding to the dynamic join point occur more than once. The developer must verify that the repetition yields correct program behaviour.

Our control flow analysis yields five disjoint categories of control flow interactions. Coupling is clearly the simplest interaction. Extension appears relatively simple, although pre- and post-conditions for the dynamic join point may require some additional analysis because dynamic join point arguments change. Narrowing interactions require analysis to understand the conditions placed by the advice on proceeding with the dynamic join point. Replacement is more complex to reason about – program invariants enforced by the dynamic join point now become the concern of the advice body. Repetition is the most complex interaction: unless the dynamic join point is idempotent, repetition requires careful examination.

$\mathbf{C} = \langle N_c, N_u, P\rangle$	**Interaction**	Rinard
$\langle 0, 1, \#\mathsf{t}\rangle$	coupling	*augmentation*
$\langle 0, 1, \#\mathsf{f}\rangle$	extension	*augmentation*
$\langle 1, 0, —\rangle$	narrowing	narrowing
$\langle 0, 0, —\rangle$	replacement	replacement
$\langle —, —, —\rangle$	repetition	*combination*

Table 3: Control Flow Interactions

Our categorization subsumes the four categories that Rinard et al. identifies, and provides a more precise distinction over the communication of dynamic join point arguments to single unconditional proceed advice. Rinard et al. provides one category: augmentation; we provide two: coupling for unchanged arguments (viz. before/after in AspectJ), and extension for modified arguments (viz. before/after in PolyAML). By inspection of Table 3, it is clear that our categorization is complete: every control interaction has a place.

5.2 Data Interaction Categories

In this section, we enumerate a covering set of state-based interactions mediated by advice. We characterize them based upon the state descriptions of the advice,

given in Equation 1 on Page 67. In this section, we use

$$A = G_A \cup S_A$$

for all the global references recognized as accessed and mutated by the advice body effect analysis and

$$P = G_P \cup S_P$$

for all the global references recognized as accessed and mutated by the pointcut effect analysis. Typically, mutation masks access, so the set of references mutated is treated as a subset of those accessed; i.e. $A = G_A$, and $P = G_P$. Our list comprises six different possible interactions, ranging from no possible interaction to very closely coupled data interactions. Just as with control interactions, we maintain Rinard et al.'s nomenclature.

Orthogonal This is where the access sets (and hence mutation sets) for the pointcut and advice body are disjoint. There is no communication between the pointcut and the advice body, other than through arguments. We say the dynamic join point and the advice are *orthogonal*. Coupling advice with orthogonal data interactions cannot influence the behaviour of the dynamic join point; allowing the programmer to understand the dynamic join point behaviour and the advice behaviour separately.

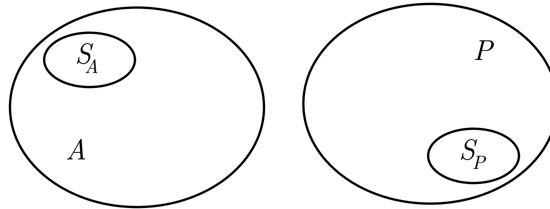


Figure 40: Orthogonal Interaction

Independent This is where the access sets may intersect, but the mutation sets remain disjoint. Although they can examine state of interest to each other, neither the pointcut nor the advice body affects the state of interest to the other. They are *independent*. Again, coupling advice with independent data interactions

allows separate understanding of the dynamic join point behaviour and the advice behaviour.

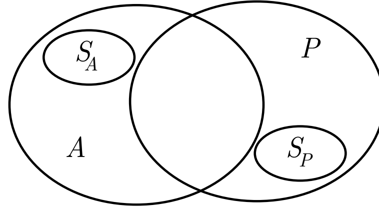


Figure 41: Independent Interaction

Observation The pointcut mutation set overlaps with the advice body's access set. This means that the advice body can *observe mutations* made during execution of the dynamic join point. With coupling advice, the behaviour of the original dynamic join point remains unchanged; but, understanding the advice behaviour does depend on understanding the dynamic join point behaviour.

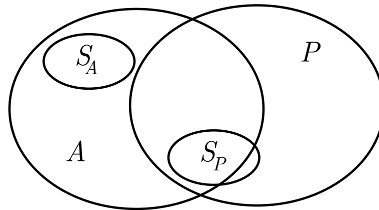


Figure 42: Observation Interaction

Actuation The advice mutation set overlaps with the pointcut's access set. This means that the dynamic join point(s) can observe mutations made by the advice body. In this case, we have a channel of communication from the advice to dynamic join point, where the advice can *actuate* the behaviour of the dynamic join point. With coupling advice, the behaviour of the advice can be understood alone; but, understanding the dynamic join point behaviour does depend on understanding the advice behaviour.

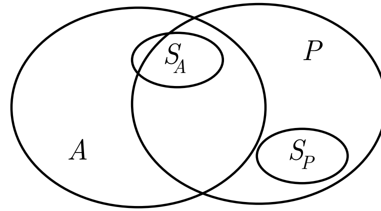


Figure 43: Actuation Interaction

Influence This occurs when the advice actuates the dynamic join point(s), and the advice observes the dynamic join point(s). That is, each has *influence* over the other by mutating a value that is accessed by the other. Except for replacement advice, this data interaction requires an understanding of the advice and the dynamic join point in order to understand the composed behaviour. Curiously, Rinard et al. omit this kind of interaction altogether – there is no category in his taxonomy that accommodates it.

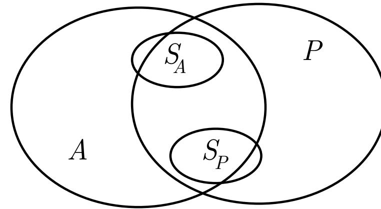


Figure 44: Influence Interaction

Interference This interaction occurs when the advice and the pointcut mutation sets overlap; that is, the dynamic join point and the advice body mutate the same field. This interaction requires both the dynamic join point behaviour and the advice behaviour to be understood and their sequencing to be understood.

Our data interactions match nicely with Rinard et al.'s taxonomy. Five of ours are identical to theirs. In our analysis, we discovered an additional category, influence, that does not appear in their categorization. By inspection of Table 4, we see that our enumeration is complete.

Rinard et al. introduces the concept of scopes, an abstraction of collections of global state. They attempt to identify these automatically by recognizing P

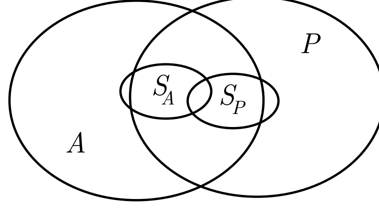


Figure 45: Interference Interaction

$A \cap P$	$A \cap S_P$	$S_A \cap P$	$S_A \cap S_P$	**Interaction**	Rinard
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	orthogonal	orthogonal
—	$\emptyset$	$\emptyset$	$\emptyset$	independent	independent
—	—	$\emptyset$	$\emptyset$	observation	observation
—	$\emptyset$	—	$\emptyset$	actuation	actuation
—	—	—	$\emptyset$	influence	*NA*
—	—	—	$\neg\emptyset$	interference	interference

Table 4: Data Interactions

and S_P sets for methods (dynamic join points). They also allow the programmer to supply an abstraction function that groups sets of state locations. Then, the data flow analysis treats any access and or mutation of a set element as involving the whole set. As this does not alter the interaction categories, we do not provide this automation.

Input (Output) Interactions

Input and output effects are informative when considered in concert with the control effect categories. This is because input and output operations are essentially not idempotent. Reading from input is a destructive operation–the data read is no longer available on the input stream. Writing to output is usually treated as un-erasable.[15] We examine four different categories of input (output) behaviour; the same analysis applies to input and output symmetrically.

15. Certainly, in some buffered situations, input and output can be undone, but entails significant effort on the part of the programmer.

$I_A(O_A)$	$I_P(O_P)$	**Interaction**
#f	#f	neither
#t	#f	advice only
#f	#t	join point only
#t	#t	both

Table 5: Input (Output) Interactions

Neither In this case, neither the advice nor the pointcut exhibit input (output) behaviour. The interaction is trivial, regardless of the control interaction.

Advice only This interaction, where the advice body performs input (output), but the pointcut effect indicates no input (output), is relatively straightforward. As the dynamic join point performs no input (output) behaviour, understanding the effect of the advice entails understanding the input (output) behaviour of the advice body only.

Dynamic join point only In this case, the pointcut effect description shows input (output) behaviour, but the advice does not. This means that understanding overall behaviour depends on whether the dynamic join point proceeds. If coupling (unconditional proceed with original arguments) occurs, then the behaviour of the dynamic join point is preserved. Extension (unconditional proceed with altered arguments) requires examination of the altered arguments to verify correct program behaviour. Replacement ensures that the dynamic join points' input (output) operation does not occur. Narrowing is more complex, because the dynamic join points' input (output) operation may or may not occur — examining the dynamic join points and the advice are important to verify program behaviour. Repetition is most complex: multiple input (output) operations will ensue as the dynamic join point is activated more than once.

Both If both the pointcut effect annotation and the advice body annotation display input (output) effects, then all cases become complex. Replacement is probably the simplest control interaction, because the dynamic join point operation is discarded. In all cases, understanding the interaction requires consideration of the dynamic join point and advice body code.

Rinard et al. do not perform input/output analysis in their classification system. Instead they treat the input/output state as a new abstract variable and group it with data-flow interactions. Our analysis is separate and provides more detail about input and output operations. As described in the previous chapter, our analysis could be sharpened to use sets of channels, rather than just boolean flags. This would lend precision to the effect region, and yield one more category, $I_A \cap I_P = \emptyset$, indicating that the two sets of channels are disjoint.

Exception Interactions

Our effect string with exceptions keeps track of the set of exceptions that might be thrown by procedures, pointcuts, and advice bodies. We also keep track of any exceptions that may be caught by *catch* blocks. This analysis allows us to recognize two anomalous situations, in addition to the usual **normal** exception propagation behaviour.

Masking The advice effect catch set overlaps with the pointcut effect throwing set. In this case, the advice is said to *mask* the exception(s) in common. This situation is highlighted for the programmer, because it is important to ensure that the exceptional circumstance is handled correctly when compared to the original handler (which we do not locate).

Injection If the advice effect throwing set is not a subset of the dynamic join point throwing set, then the advice is said to *inject* a new exception into the control flow. This situation is highlighted for the programmer because it is possible that it is not handled at a higher level in the program. This would result in advice creating un-handled exception errors.

Rinard et al. do not categorize exception behaviour, except to insist that augmenting interactions must not throw exceptions. This may be a result of using AspectJ as the testbed for their system. In comparison to checked exceptions in Java, our injection analysis appears to add no additional value. The Java language expects all checked exceptions to be annotated and verified at compilation time. Unfortunately, the Java virtual machine specification does not encompass checked-exception annotations and verification. Therefore, aspects in AspectJ, when compiling to bytecode, can inject exceptions into program behaviour with-

out the usual Java compilation warnings. This flexibility is valuable, but marks a strong distinction between Java and AspectJ.

Concurrency Interactions

Our concurrency analysis is simple. Our effect analysis yields a pair of effect strings, $E = \langle E_{\text{sync}}, E_{\text{async}} \rangle$, one describing the behaviour in the original thread (E_{sync}) and a second separate effect string that combines effects from forked threads (E_{async}). This separation allows us to recognize which effects will occur asynchronously. We recognize three categories of concurrency, by examining the the content of each effect string:

Synchronous In the synchronous case, there are no effects visible in the E_{async} effect string. This is the most common case, and the simplest interaction to understand: the sequential ordering of the program is preserved and described by the kind of control interaction given for E_{sync}.

Mixed In this case, neither E_{sync} nor E_{async} are empty. This is the most complex interaction, and highlights to the programmer that the dynamic join points and the advice need to be examined.

Asynchronous In the asynchronous case, there are no effects visible in the E_{sync} effect string. This case is not as simple as the synchronous case, because we conservatively combined all spawned threads together. The programmer must carefully examine the code to understand the parallelism and potential race conditions.

Our analysis does allow the programmer to identify the following example situations:

1. an advice which pushes push an exception into another thread, and out of the control flow of an exception handler, these exceptions will never be caught by the original handler.

2. an advice which pushes state mutation into a new thread: any expectation of atomicity arising from default synchronous behaviour is unwarranted.

3. an advice which pushes input or output into a new thread also waives any expectation of sequential behaviour.

The programmer can be directed to those advice bodies which are are highlighted as bringing *asynchrony* effects.

Summary of Interactions

To summarize, our effect analysis, developing as a natural static analysis from our continuation-based dynamic semantics, yields a classification system for aspect-oriented programs. It is very similar to Rinard et al., but more precise and complete.

1. It offers more precise control interaction categories, in particular, differentiating two components of Rinard's augmentation class into coupling and extension classes,

2. It is complete with regard to data interaction categories, by including the influence class which is missing in Rinard's categorization,

3. It characterizes exceptions and concurrency in a simple way, but highlights interactions that are omitted by Rinard,

4. It has allowed us to identify an inconsistency between Java and AspectJ, where checked exceptions may be introduced by advice without triggering the usual Java errors.

5.3 Compound Interactions

Compound interactions occur when two (or more) advice compose at a single dynamic join point. To preserve deterministic behaviour, one advice must go first, and the next advice applies only when invoked when the first one proceeds. This is similar to applying advice at advice-execution dynamic join points. The overall effect of the two advice is to layer the effect of the dominant one over the subordinate one.

Purely from a containment perspective, we report when multiple advice have common pointcuts, using whole-program analysis. That is, when two advice have an overlapping set of dynamic join points, we draw the programmer's attention to this. For example, in the following code, we indicate that *adv12* and *adv13* affect the same dynamic join points, namely calls to *p1*. We draw the programmer's

```
(((p1 proc () 1)
  (p2 proc () 2)
  (p3 proc () 3)
   (adv1 advise (call p1 ())
         ...)
   (adv12 advise (or (call p1 ())
                     (call p2 ()))
           ...)
   (adv23 advise (or (call p2 ())
                     (call p3 ()))
           ...))
 ...)
```

Figure 46: Multiple Advice at Dynamic Join Points

attention to the fact that both advice apply; it remains her duty to ensure they work together.

The significant consideration is that the order of application matters. Besides textual ordering, another natural order is to consider pointcuts as subsetting dynamic join points. In Figure 46, the set of dynamic join points matched by the advice *adv1* is a subset of those matched by *adv12*. Therefore, *adv1* can be considered as more-specific, since it affects a smaller set of dynamic join points. Nelson et al. [123] examined a variety of orderings for a simple bounded buffer system, and showed that most-specific to least-specific offers the best result – it maintained more liveness properties. AspectJ adopts this most- to least- specific strategy. In some applications, this ordering does not provide the correct semantics; hence **declare precedence** (superseding the previous **dominates** construct) was added.

This ordering is not generally computable; as the example in Figure 46 shows. Advice *adv12* is neither more specific nor less specific than *adv23*. In this case, some other criterion must be used. During execution, lexical ordering is applied. The programmer may not anticipate this ambiguity; so the default ordering can be highlighted in the effect report.

For these reasons, we simply supply the effect categorization, and report the advice ordering and conflicts to the programmer. It remains their responsibility to determine whether the layering provides the desired results.

```
(run '(([depth global]
        [f proc (x) (if (call = 0 x)
                        (begin (call display (get depth))
                               1)
                        (call * x (call f (call - x 1)))))]
        [bef advise (exec f v) (begin (call display (get depth))
                                      (set depth (call + (get depth) 1))
                                      (proceed v))]
        [aft advise (exec f v) (let ([r (proceed v)])
                                 (begin (call display (get depth))
                                        (set depth (call - (get depth) 1))
                                        r))])
       (begin (set d 0)
              (call f 5))))
```

Figure 47: Tracing Instrumentation

5.4 Example Interactions and Reports

Here, we provide the results of applying our analysis to four archetypical AOP programming situations.

Tracing

The tracing aspect given in Section 2.4.2 of Laddad [98] instruments a given method call with before and after logging messages. We translate this to our extended Proc language, yielding the code displayed in Figure 47.

Our analysis of this instrumentation code shows that these advice have the following interaction behaviours.

Each of the advice interact with the dynamic join point as

- **coupling** – the arguments are passed through, unchanged, to the single unconditional proceed;
- **actuation** – the *fact* procedure reads the *depth* global value, which is mutated by each advice,
- **advice output** – the advice body provides additional output effects;

```
(run '(([minx global]
        [xpos global]
        [a1 advise (set xpos v) (if (call < v (get minx))
                                    (raise outOfBounds)
                                    (proceed v))]
        [a2 advise (set minx v) (if (call < (get xpos) v)
                                    (raise outOfBounds)
                                    (proceed v))])
       (begin (set minx 1)
              (set xpos 2)
              (set xpos 1)
              (set xpos 0))))
```

Figure 48: Move Limiting

- **normal exceptions** – no addition masking or injection of exceptions occurs;
- **synchronous** – no effects are pushed into additional threads.

Furthermore, each advice interacts with the other at the same dynamic join point with **interference** state effects, because each reads and writes the *depth* global. Our report indicates that the advice apply to the same dynamic join points, and so they conflict. This warns the programmer to review their ordering (which is lexical in our implementation).

Move Limiting

From Clifton and Leavens [28], we take the `MoveLimiting` aspect, where mutation of a state value is restricted to positive values by an aspect. Our translation is shown in Figure 48.

Our analysis against bounds checking code shows that each advice has the following interaction behaviour:

- **coupling** – the arguments are passed through, unchanged, to the single unconditional proceed;

```
(run '(([p proc (x) (if (call = 0 x)
                        (raise zero)
                        1)]
        [a advise (exec p v) (try (proceed v)
                                  catch zero (begin (call display 0)
                                                    (raise exc)))])
       (begin (call p 1)
              (call p 0))))
```

Figure 49: Exception Logging Instrumentation

- **orthogonal** – there is no interaction regarding state variables;
- **none** – the advice body provides no additional input/output behaviour;
- **injects exceptions** – it injects a new exception into the flow control of the program, and the programmer should be notified to ensure that appropriate *catch* expressions are in place;
- **synchronous** – no effects are pushed into additional threads.

This advice ensures that *xpos* is never less than *minx*. There is no state interaction – *minx* is not referenced by *set xpos* expressions, and vice versa. However, there is notification to the programmer that a new exception is injected into the program. It becomes their responsibility to ensure the correct *catch* expressions are available.

Exception Logging

The exception logging advice given in Section 5.4.2 of Laddad [98] instruments an application with after advice that logs exceptions thrown from method invocation. We translate this to our extended Proc language, yielding the code displayed in Figure 49.

Our analysis against simple exception throwing code shows that this advice has the following interaction behaviour:

- **coupling** – the arguments are passed through, unchanged, to the single unconditional proceed;

```
(run '(([ret global]
        [p proc (x) (set ret x)]
        [a advise (exec p v) (fork (proceed v)
                                   0)])
       (begin (set ret 0)
              (call p 1)
              (call (display (get ret)))
              (set ret 2)))
              (call p 0))))
```

Figure 50: Runnable With Return

- **orthogonal** – neither reading nor writing of mutable references occurs in the advice body;
- **advice output** – the advice body provides additional output effects;
- **normal exceptions** – no addition masking or injection of exceptions occurs;
- **synchronous** – no effects are pushed into additional threads.

This advice couples the instrumentation code to exception propagation out of the named procedure. Our analysis shows that it can be reasoned about modularly, in isolation from the rest of the program.

Runnable With Return

Laddad [98] provides a number of examples where execution is deferred to another thread, using his `RunnableWithReturn` class. Here we provide a simple implementation of this for Proc in Figure 50, and apply our effect analysis to it.

The system reports the effect of the dynamic join point (*exec p x*) as mutating the global variable *ret*. Although this is a simple example, that summary can alert the programmer that *ret* is also updated outside of *p*, as in the main body of the program.

The key result from our analysis is to note that this code inserts an **asynchronous** effect into the program. In particular, setting the global *ret* via the

procedure p is asynchronous and can interfere with other uses of *ret*. The programmer is notified of the potential race condition.

With this understanding of the analysis and its use, we turn our attention to the sorts of analyses that others have recently proposed.

5.5 Other Analyses and Related Work

Other research groups have recognized the effect of advice on programming. Clifton and Leavens [27, 28] have provided another taxonomy of spectators, observers, and assistants. Katz [89]; Katz and Gil [90]; Sihman and Katz [149] has explored a third set of descriptions for understanding the effects of pointcuts and advice. Dantas and Walker [40] provide a type-and-effect characterization of harmless advice.

Clifton et al.

Clifton proposes two distinct kinds of advice interaction: based on the effect of the advice on the dynamic join point. They are called spectators and assistants. Here, we examine each of these, and place it as a subset of our interaction taxonomy.

Spectators Spectators, originally termed observers, are aspects which do "not change the behaviour of any other module." Specifically, a spectator

- may only mutate state that it owns (i.e. not accessed by other modules)
- may mutate the program world state (i.e. perform input/output),
- must not change the control flow to or from the dynamic join point,
- must proceed exactly once, unconditionally and without changing arguments,
- must not explicitly throw any checked exceptions,

Therefore, spectators are aspects with coupling and observation interactions and with no exception or concurrency interactions. They are, by design, simple to reason about – the advice and the dynamic join point behaviour can be understood and implemented as separate modules.

Assistants Assistants are the catch-all for any other type of aspect; those that change the behaviour of the advised module. These interactions encompass all other kinds in our taxonomy.

The goal of Clifton et al.'s work is to identify advice whose interactions are simple enough to be understood independently. They provide a static analysis for a substantial object-oriented language, including the necessary alias and pointer analysis, to ensure that spectators conform to the specification. They syntactically enforce coupling interaction by marking spectators as `before`, `after`, or `surround` to ensure the unconditional, single proceed with original arguments. Any other kind of advice is an assistant, and they insist that modules include explicit reference, an `accepts` declaration, for any permitted assistant. In this way, their system provides a measure of modular reasoning: either an advice is a spectator and has no impact on reasoning about any module, or it is an assistant and must be explicitly documented in the modules with which it interacts.

Katz et al.

Katz [89] together with Katz and Gil [90] have categorized aspects based on temporal properties. They identify three main categories of aspects: spectative, regulative, and invasive, based upon a sophisticated state graph model. In particular, they characterize aspects as additive, in that they extend an underlying program – one without aspects. Here, we show how their categories are subsumed by our taxonomy.

Spectative A spectative aspect "can change the values of variables local to the aspect, but does not change the value of any variable or the flow of method calls of the underlying system." This is directly characterizes as providing a refinement of the underlying state graph – inserting state nodes with effects not shared with the unadvised system. This description matches the coupling and observation interactions that we previously described.

Katz [88] relaxes these restrictions to permit termination of execution as well. These are called *weakly spectative*. We do not have a specific category for this possibility; it seems to be narrowing or replacement, but with a very specific alternative operation.

Regulative A regulative aspect is a spectative aspect that is permitted to remove state graph edges without adding connections, and to repeat some states nodes. Removing state graph edges without adding connections corresponds with our narrowing and our replacement control interactions. Repeating a state note does not include multiple proceeds, because that adds additional connections. Furthermore, repeating a state node is permitted only when the repeated node does not alter the state other than that local to the aspect. So, regulative aspects must be observational.

Invasive Invasive aspects are permitted to alter state graph of the underlying system in any way. This catch-all group contains the remaining interaction possibilities in our classification.

In principle, invasive aspects can invalidate any property of the system; but Katz et al. claim that many invasive aspects change the underlying system in restricted ways. Therefore, they separate out a group of *weakly invasive* aspects — ones which augment the state graph with new transitions beginning only at points in the underlying state graph. They leverage this to show that a property of the original system is preserved if the advice body also preserves the property, without checking the entire system. They indicate that deriving this property is not possible statically, but the examples they give fit into the extension+observation and coupling+actuation groups.

Dantas et al.

Dantas and Walker [40] provide a type-and-effect analysis for harmless advice – that which obeys a weak non-interference property. Specifically, harmless advice

- may change the termination behaviour of computations,
- may perform input/output operations,
- but does not otherwise influence the final result.

This allows programmers to ignore harmless advice when reasoning about the partial correctness of their programs, and to extend programs without breaking important data invariants offered by the original program. They recognize aspects providing profiling, invariant checking, security enforcement, instrumentation, and persistence features as harmless. Although their analysis system differs from

Rinard et al., they declare that harmless advice falls into orthogonal, independent, and observation interactions.

Some of this work takes effect descriptions in different direction, into the checking realm. In particular Clifton et al. insists that every assistant be visible to any module that it affects, by including its name in a new `accepts` declaration. Failing to annotate this implies that the aspect must be a spectator, and Clifton provides static analysis to prove this property. Failure of this check causes compilation errors.

Krishnamurthi et al. [97] provide a different model-checking system for aspects. They expect the programmer to supply computational tree logic descriptions of desired properties of the system and apply a model checker to prove these of a system augmented with aspects.

5.6 Summary

Our effect analysis from Chapter 4 has given us the information to provide an automated system of classifying advice and highlighting interactions that merit programmer inspection. Our classification system, based upon interaction and comparisons of the effect strings, refines that of Rinard et al. by

- subdividing the augmentation control interaction into *extension* and *coupling*
- recognizing a missing data interaction, *influence*,
- providing an alternate input/output characterization that can be extended to recognize multiple I/O channels,
- includes exception interactions in a more general way, and
- provides simple concurrency categories.

Our exception categorization also led us to discover an unexpected property of AspectJ, namely that it does not provide the same assurances as Java that all checked exceptions are annotated in method signatures. Last, we have shown how our categorization also subsumes those of Clifton et al., Katz (et al.), and Dantas et al.

Our classification could become the foundation for annotations that express desired advice interactions and for statically checking effects against those annotations. Those goals are beyond the scope of this work. Our success is to show that our semantic development provides a workable effect analysis system.

CHAPTER 6

Conclusion

Having presented the technical material of our work, we review the contributions of this dissertation and close with a summary of open research questions arising from our results.

6.1 Contributions

This research provides two semantic descriptions of dynamic join points, pointcuts, and advice for procedural languages.

One, a dynamics semantics, moves from our previously published expression-oriented, big-step system to a novel continuation-based, small-step semantics. This translation yields an elegant model of dynamic join points as principled program control points, pointcuts as identifiers of these points, and advice as specializers of the behaviour of these control points. The second semantic specification, a static semantics, captures the essential abstraction of continuations, that of computational effects, and develops an abstraction of pointcuts and advice with regard to the effects they express. This abstraction to effects supports and refines existing aspect classifications, yielding interesting types-and-effects properties for dynamic joinpoints, pointcuts, and advice.

The specific contributions are:

1. A novel development of continuation-based dynamic semantics for dynamic join points, pointcuts, and advice for a first-order, mutually-recursive pro-

cedural language showing that

a) Dynamic join points, pointcuts, and advice aspects can be modeled directly in continuation semantics; without the need for labels or continuation marks,

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

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

2. An application of this construction to a higher-order procedural language, Scheme, yielding a semantic description of AspectScheme which includes lexically-scoped and dynamically-scoped pointcuts and advice.

3. An implementation of AspectScheme, constructed as a language extension to PLT Scheme, using macros and their language extension points, to supply, and lexically-scoped, dynamically-scoped, and the more usual top-level (declarative) pointcuts and advice aspects.

4. A demonstration that *cflow* pointcuts break tail-call properties of programming languages and add a state effect into the languages.

5. A static semantics that focusses on the key property of continuations: that they carry computational effects. We

a) characterize dynamic join point shadows by their input, output, state access and state mutation regions;

b) associate dynamic join point effects with pointcuts, yielding reports summarizing and contrasting these effects;

c) characterize advice bodies, describing their input/output and state effects as well as any repetition of join point behaviour.

6. An effect reporting algorithm that extends ones already accepted for aspect-oriented languages. Ours includes

a) five control interaction classes that cover a broader range,

b) six data interactions, including one missing from the existing analyses,

c) an alternate input (output) categorization with four categories,

d) exception categorization that helped highlight an AspectJ/Java inconsistency, and

e) three simple concurrency interactions.

Any substantial research must focus on specific questions, leaving others for future work. In addition, as it solves some problems, it must illuminate new areas of investigation also. We now consider some of these.

6.2 Open Questions

As semantics for AOP languages is still in its infancy, many unanswered questions remain. Even with its limited focus on dynamic aspects, this work leaves open many avenues for further investigation. They come in three clusters: one related to directly extending and further formalizing this work; one related to alternate language families – specifically object-oriented languages; and one related to productizing the effect analysis.

Extending and Formalizing

Implementation Our construction provides an elegant account of dynamic join points, pointcuts, and advice. The efficiency of this model, however, is unclear. Tail call optimization has been preserved, at the cost of exposing *cflow*'s internal effects. Exposing administrative frames which separate operator and operand reduces the potential for optimal instruction reordering. CPS optimizations identified by Shivers [148] and others may be invalidated, and partial evaluation opportunities given by Danvy et al. [43] and Damian and Danvy [39] may become unavailable.

Full Abstraction Several semantic specifications of dynamic join points, pointcuts, and advice aspect-oriented languages have been posed[20; 42; 58; 115; 169]. This work identifies the underlying continuation structure which this kind of AOP attempts to abstract and modularize. The various specifications differ in subtle ways: some provide syntactic control to enforce coupling interactions (`before`, `after`, `surround` advice types); some relax this to extension interactions; others

expand beyond declarative pointcut languages. How can we be sure that we've captured the essence of AOP?

We attempt to do this by construction from the denotational semantics of our previous work[169]. By virtue of the correctness of the CPS conversion, our abstract machine semantics matches the original at relevant points, but it is more precise at the auxiliary continuations. This over-specification is undesirable.

Formal verification of the equivalence of language semantics is a *full abstraction problem*: show that two models are observationally equivalent – neither is more expressive than the other. Providing this level of correspondence between abstract machine and denotational semantics for procedural languages was challenging, and the complete result is relatively recent [3]. Sub-typing has only recently become expressible in game semantics, so a full OO language with dynamic dispatch seems still to be a ways off.

Solving the full abstraction problem for procedural languages required a new kind of semantic specification: (two-player) game semantics [2]. Recently, Abramsky [1] has extended this theory to multi-player game semantics, which I believe offers the right framework for full abstraction of AOP. Otherwise, one must choose to compose aspects either into the language (i.e., extending the opponent) or into the program (i.e., rewriting the player).

Inter-type Declarations Masuhara and Kiczales [112] demonstrate that the pointcuts and advice model generalizes a number of other AOP forms.

- Open classes [29] and inter-type declarations [23]: in this case, we posit a dynamic semantics for the elaboration phase [21][16].
- Composition filters [13] and HyperJ [128]: both of which are based on a domain-specific program composition language.
- Traversals as given in Demeter [107]: which is based on a traversal definition language.

In our view, each of these languages have a (well-defined) dynamic semantics, amenable to CPS conversion and defunctionalization. We see each as providing a set of dynamic join points, and a method for composing new behaviour at those

16. For example, Scheme R^6RS standardization work has specifically recognized this in the `letrec*` formulation of `define`.

dynamic join points. For example, inter-type declarations appear interested in side-effecting elaboration-stage values such as methods, classes, and operations. Is this construction truly as general as it appears?

Type-checking Aspects Our work explicitly eschews type checking. There has been some work on applying type-checking and type-inference to aspects [41; 42]. Type inference of `around` advice is especially difficult; it is conjectured [165] to be impossible in pure Hindley-Milner-Damas [38; 127; 150; 158]. Recently, Dantas et al. [42] provide a clever and novel blending of global and local type inference [132] for parametric polymorphism in aspects. The scheme is still impredicative, and requires some type annotations. Interesting remaining questions include “how much type annotation is required?”, “are there relationships between advice that requires annotation and the effects it provides?”, and “can aspects replace type constraints across data flows to permit greater modularity?”.

Object-Oriented Languages

Our focus has been on procedural languages – systems providing alternative dimensions of modularity. We perceive a cluster of open questions revolving around aspects and object languages.

Dynamic Join Point Construction We believe that that our semantic construction will yield the same intuitive dynamic join points that the original aspect language designers identified. An initial implementation looks promising.

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

Figure 51: Object-Oriented Dynamic Join Points

Object / Aspect Duality Filinski [65] noted the categorical duality between values and continuations. Object-oriented technology has provided abstract and modular values. What might the equivalent abstraction and modularity of continuations be? How might the Galois connection between direct and CPS semantics, explored by Danvy [44] and Danvy and Lawall [49], affect this modularity relationship? We hypothesize that the result is similar to pointcuts and advice AOP. If so, that helps us understand a fundamental question: *what do aspects modularize*?

An Effect Checking Tool

We give an effect reporting tool. However, effects can compose in a variety of ways. Different orderings give different behaviours, such as resetting state on exceptions (yielding transactional behaviour), or preserving state on exceptions (the more common behaviour). Both of these are legitimate; therefore, an effect-checking tool requires a syntax for annotating desired effects. One place where these kinds of properties are again being investigated is in *typestate checking* [64; 101; 156] and *regular types* [76; 126].

Other than in research prototypes for domain-specific applications (e.g. concurrency [7] and mobility [96]), effects are most common as monadic type annotations in Haskell and related languages. Perhaps these syntaxes can be adapted for use with AOP, to describe layered and combined effects like transactions. This is an open question, because making it expressive and lightweight are opposing forces: utility is the desired end – usability studies seem to be mandated. Further, this composition is problematic [60; 85; 95; 106]. Three ways to circumvent these composition difficulties are

- to examine our construction as a special case of delimited continuations [15; 16; 87; 145; 161] where each frame is captured by *shift* and *reset*. In future work, we intend to examine this relationship as a degenerate form of monadic reflection, following Shan's lead in applying polarized logic to help illuminate the construction [146];
- to re-examine our construction in terms of monads, potentially yielding the sort of configurable applications of Angus [9];
- or to examine a more mathematical formulations of effects, such as Führmann [75].

Works Cited

[1] Samson Abramsky. Socially responsive, environmentally friendly logic. Oxford University Computing Laboratory, 2006.

[2] Samson Abramsky and Radha Jagadeesan. Games and full completeness for multiplicative linear logic. *Symbolic Logic*, 59(2):543–574, 1994.

[3] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for pcf. *Information and Computation*, 163(2):409–470, 2000.

[4] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Principles and Practice of Declarative Programming*, pages 8–19. ACM Press, August 2003. ISBN 1-58113-705-2.

[5] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005.

[6] R. E. Allen, H. W. Fowler, and F. G. Fowler, editors. *The Concise Oxford Dictionary*. Oxford University Press, 8th edition, 1990.

[7] Torben Amtoft, Flemming Nielson, and Hanne Riis Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, June 1999.

[8] James H. Andrews. Process-algebraic foundations of aspect-oriented programming. In Yonezawa and Matsuoka [170], pages 187–209. ISBN 3-540-42618-3.

[9] Chris Angus. Constructing configurable applications by combining monads. Manuscript, 1997. URL `http://www.cs.ncl.ac.uk/old/research/trs/paper/577.ps`.

[10] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN 0-521-41695-7.

[11] Awe Aßmann and Andreas Ludwig. Aspect weaving by graph rewriting. In U.W. Eisenecker and K. Czarnecki, editors, *Generative Component-based Software Engineering*, October 1999.

[12] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sasha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An Extensible AspectJ Compiler. In Mehmet Akşit, editor, *Aspect Oriented Software Development*, pages 87–98. ACM Press, April 2005.

[13] Lodewijk Bergmans and Mehmet Akşit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001.

[14] Edoardo Biagioni, Ken Cline, Peter Lee, Chris Okasaki, and Chris Stone. Safe-for-space threads in standard ML. *Higher-Order and Symbolic Computation*, 11(2):209–225, 1998.

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

[16] Dariusz Biernacki, Olivier Danvy, and Chung chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60(3):274–297, 2006.

[17] Bruno Blanchet. Escape analysis for java: Theory and practice. *Transactions on Programming Languages and Systems*, 25(6):713–775, 2003.

[18] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In Murphy and Lieberherr [121], pages 83–92.

[19] Rodney A. Brooks, Richard P. Gabriel, and Guy L. Steele, Jr. An optimizing compiler for lexically scoped LISP. In *Compiler Construction*, pages 261–275, 1982.

[20] Glenn Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μABC: A minimal aspect calculus. In Philippa Gardner and Nobuko Yoshida, editors, *Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224. Springer-Verlag, September 2004. ISBN 3-540-22940-X.

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

[22] Luca Cardelli, editor. *European Conference on Object Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, July 2003. Springer-Verlag. ISBN 3-540-40531-3.

[23] Andy Clement, Adrian Colyer, George Harley, and Matthew Webster. *Eclipse AspectJ*. Addison-Wesley, 2005. ISBN 0321245873. Chapter 8.

[24] John Clements and Matthias Felleisen. A tail-recursive semantics for stack inspection. In *Lecture Notes in Computer Science*, volume 2618, pages 22–37, July 2003.

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

[26] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[27] Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. Technical Report TR#02-04a, Iowa State University, April 2002.

[28] Curtis Clifton and Gary T. Leavens. Spectators and assistants: Enabling modular aspect-oriented reasoning. Technical Report TR#02-10, Iowa State University, October 2002.

[29] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, volume 35(10), pages 130–145, 2000.

[30] William D. Clinger. Proper tail recursion and space efficiency. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, June 1998.

[31] Yvonne Coady, Gregor Kiczales, Mike Feeley, Norm Hutchinson, and Joon Suan Ong. *Structuring Operating System Aspects*, chapter 28, pages 651–657. In Filman et al. [71], October 2004.

[32] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[33] Patrick Cousot and Radhia Cousot. Galois connection based abstract interpretations for strictness analysis. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 98–127. Springer-Verlag, June 1993. ISBN 3-540-57316-X.

[34] Patrick Cousot and Radhia Cousot. Temporal abstract interpretation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–25, 2000.

[35] Patrick Cousot and Radhia Cousot. Modular static program analysis. In R. Nigel Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 159–178. Springer-Verlag, April 2002. ISBN 3-540-43369-4.

[36] Patrick Cousot and Radhia Cousot. Basic concepts of abstract interpretation. In René Jacquart, editor, *International Federation for Information Processing Congress*, pages 359–366. Kluwer, August 2004. ISBN 1-4020-8156-1.

[37] Radhia Cousot, editor. *Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, January 2005. Springer-Verlag. ISBN 3-540-24297-X.

[38] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[39] Daniel Damian and Olivier Danvy. Syntactic accidents in program analysis: On the impact of the cps transformation. September 2003.

[40] Daniel S. Dantas and David Walker. Harmless advice. In Peyton Jones [130], pages 383–396. ISBN 1-59593-027-2.

[41] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. PolyAML: A polymorphic aspect-oriented functional programming language. In Danvy and Pierce [51]. ISBN 1-59593-064-7.

[42] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, to appear.

[43] O. Danvy, B. Dzafic, and F. Pfenning. On proving syntactic properties of cps programs. In Andrew Gordon and Andrew Pitts, editors, *Workshop on Higher Order Operational Techniques in Semantics*, September 1999. ENTCS, vol. 26.

[44] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22 (3):183–195, 1994.

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

[46] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Lisp and Functional Programming*, pages 151–160, 1990.

[47] Olivier Danvy and John Hatcliff. Thunks (continued). In *Workshop on Static Analysis*, pages 3–11, 1992.

[48] Olivier Danvy and John Hatcliff. On the transformation between direct and continuation semantics. In Stephen D. Brookes, Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 627–648. Springer-Verlag, April 1993. ISBN 3-540-58027-1.

[49] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. pages 299–310.

[50] Olivier Danvy and Lasse R. Nielson. A first-order one-pass cps transformation. *Theoretical Computer Science*, 308(1–3):239–257, November 2003.

[51] Olivier Danvy and Benjamin C. Pierce, editors. *International Conference on Functional Programming*, September 2005. ISBN 1-59593-064-7.

[52] Bart De Win, Wouter Joosen, and Frank Piessens. *Developing Secure Applications Through Aspect-Oriented Programming*, chapter 27, pages 633–560. In Filman et al. [71], October 2004.

[53] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In *Lecture Notes in Computer Science*, volume 2192, pages 170–186, September 2001.

[54] Christopher J. Dutchyn. AspectScheme v1. PLaneT repository, January 2005.

[55] Christopher J. Dutchyn. AspectScheme v2. PLaneT repository, January 2006.

[56] Christopher J. Dutchyn, Gregor Kiczales, and Hidehiko Masuhara. Aspect Sandbox. internet, 2002. URL `http://www/labs/spl/projects/asb.html`.

[57] Christopher J. Dutchyn, Hidehiko Masuhara, and Gregor Kiczales. AOP language exploration using the aspect sand box. In Harold Ossher and Gregor Kiczales, editors, *Aspect Oriented Software Development*. ACM Press, April 2002. ISBN 1-58113-469-X. Tutorial.

[58] Christopher J. Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 67(3):207–239, November 2006.

[59] Matthew B. Dwyer and Richard N. Taylor, editors. *Foundations of Software Engineering*, November 2004. ACM Press. ISBN 1-58113-855-5.

[60] David Espinosa. Building interpreters by transforming stratified monads. Unpublished manuscript, June 1994.

[61] Matthias Felleisen. The theory and practice of first-class prompts. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 180–190, 1988.

[62] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102: 235–271, 1992.

[63] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full jumps. In *Lisp and Functional Programming*, pages 52–62, 1988.

[64] John Field, Deepak Goyal, G. Ramalingam, and Eran Yahav. Typestate verification: Abstraction techniques and complexity results. *Science of Computer Programming*, 58(1–2):57–82, 2005.

[65] Andrzej Filinski. Declarative continuations and categorical duality. Master's thesis, DIKU, University of Copenhagen, August 1989.

[66] Andrzej Filinski. Representing monads. In Hans J. Boehm, Bernard Lang, and Daniel Yellin, editors, *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457. ACM Press, January 1994.

[67] Andrzej Filinski. *Controlling Effects*. PhD thesis, University of Pennsylvania, May 1996.

[68] Andrzej Filinski. Representing layered monads. In Andrew Appel and Alex Aiken, editors, *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 175–188. ACM Press, January 1999.

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

[70] Robert Filman and Daniel Friedman. *Aspect-Oriented Programming is Quantification and Obliviousness*, chapter 2, pages 21–36. In Filman et al. [71], October 2004.

[71] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, October 2004.

[72] Robert B. Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

[73] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237–247, 1993.

[74] Daniel Friedman, Mitchell Wand, and Christopher Haynes. *Essentials of Programming Languages*. MIT Press, 2001.

[75] Carsten Führmann. Varieties of effects. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 144–159. Springer-Verlag, 2002.

[76] Vladimir Gapeyev and Benjamin C. Pierce. Regular object types. In Cardelli [22], pages 151–175. ISBN 3-540-40531-3.

[77] K. M. George, Janice Carroll, and Dave Oppenheim, editors. *Control Flow Analysis: a Compilation Paradign for Functional Language*, February 1995. ACM Press. ISBN 0-89791-658-1. revised version online.

[78] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[79] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck. Handling crosscutting constraints in domain-specific modeling. *Communications of the ACM*, 44(10):87–93, 2001.

[80] William G. Griswold and Mehmet Akşit, editors. *Aspect Oriented Software Development*. ACM Press, March 2003. ISBN 1-58113-660-9.

[81] Williams Ludwell Harrison, III. *The Interprocedural Analysis and Automatic Parallelization of Scheme Programs*. PhD thesis, University of Illinois at Urbana-Champagne, 1989.

[82] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 458–471, 1994.

[83] Paul Hudak, Simon Peyton Jones, and Phil Wadler. Report on the programming language Haskell: a non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5), May 1992. Version 1.2.

[84] Erik Hilsdale and Jim Hugunin. Advice Weaving in AspectJ. In Murphy and Lieberherr [121], pages 26–35.

[85] Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report DCS/RR-1004, Yale University, 1993 1993.

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

[87] Y. Kameyama. Towards logical understanding of delimited continuations. In A. Sabry, editor, *Continuations Workshop*, number 545, pages 27–33. Computer Science Department, Indiana University, 2000.

[88] Shmuel Katz. Diagnosis of harmful aspects using regression verification. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *Foundations of Aspect Oriented Languages*, March 2004.

[89] Shmuel Katz. Aspect categories and classes of temporal properties. *Lecture Notes in Computer Science*, 3880:106–134, 2006.

[90] Shmuel Katz and Yossi Gil. Aspects and superimpositions. In Ana M. D. Moreira and Serge Demeyer, editors, *European Conference on Object Oriented Programming*, volume 1743 of *Lecture Notes in Computer Science*, pages 308–309. Springer-Verlag, June 1999. ISBN 3-540-66954-X.

[91] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[92] Gregor Kiczales. The fun has just begun. In Griswold and Akşit [80]. invited talk.

[93] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, 2001.

[94] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, June 1997.

[95] David King and Philip Wadler. Combining monads. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 134–143. Springer-Verlag, July 1993. ISBN 3-540-19820-2.

[96] Zeliha Dilsun Kirli. *Mobile Computation with Functions*. Advances in Information Security. Kluwer, 2002. ISBN 1-4020-7024-1. also LCS Edinburgh dissertation, 2001.

[97] Shriram Krishnamurthi, Kathi Fisler, and M. Greenberg. Verifying aspect advice modularly. In Dwyer and Taylor [59], pages 137–146. ISBN 1-58113-855-5.

[98] Ramnivas Laddad. *AspectJ in Action*. Manning Press, April 2003. ISBN 1930110936.

[99] Patrick Lam, Victor Kuncak, and Martin Rinard. On modular pluggable analyses using set interfaces. Technical Report 933, MIT, 2003.

[100] Patrick Lam, Victor Kuncak, and Martin Rinard. On our experience with modular pluggable analyses. Technical Report 965, MIT, 2004.

[101] Patrick Lam, Viktor Kuncak, and Martin C. Rinard. Generalized typestate checking for data structure consistency. In Cousot [37], pages 430–447. ISBN 3-540-24297-X.

[102] Peter J. Landin. A generalization of jumps and labels. UNIVAC Systems Programming Research Report, August 1965. Reprinted in *Higher-Order and Symbolic Computation*, 11(2):125–143, 1998.

[103] Gary T. Leavens and Ron Cytron, editors. *Foundations of Aspect Oriented Languages*, April 2002. Iowa State University TR#2-06.

[104] Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In Jean-Yves Girard, editor, *Typed Lambda Calculus and Applications*, volume 1581 of *Lecture Notes in Computer Science*, pages 228–242. Springer-Verlag, April 1999. ISBN 3-540-65763-0.

[105] Sheng Liang. *Modular Monadic Semantics and Compilation*. PhD thesis, Yale, 1997.

[106] Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, 1995.

[107] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10): 39–41, 2001.

[108] Karl J. Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Transactions on Programming Languages and Systems*, 26(2):370–412, 2004.

[109] Daniel Lohmann and Olaf Spinczyk. Architecture-neutral operating system components. In Michael L. Scott and Larry Peterson, editors, *Symposium on Operating Systems Principles*. ACM Press, October 2003. work-in-progress session.

[110] J. M. Loucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. PhD thesis, MIT, 1987. LCS/TR-408.

[111] J. M. Loucassen and D. K. Gifford. Polymorphic effect systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 47–57. ACM Press, 1988.

[112] Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In Cardelli [22], pages 2–28. ISBN 3-540-40531-3.

[113] Hidehiko Masuhara, Gregor Kiczales, and Christopher J. Dutchyn. Compilation semantics of aspect-oriented programs. In Leavens and Cytron [103], pages 17–26.

[114] Hidehiko Masuhara, Gregor Kiczales, and Christopher J. Dutchyn. A compilation and optimization model for aspect-oriented programs. In Görel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, January 2003.

[115] Hidehiko Masuhara, Hideaki Tatsuzawa, and Akinori Yonezawa. Aspectual caml: an aspect-oriented functional language. In Danvy and Pierce [51], pages 320–330. ISBN 1-59593-064-7.

[116] Albert R. Meyer and Jon G. Riecke. Continuations may be unreasonable. In *Lisp and Functional Programming*, pages 63–71, 1988.

[117] Matthew Might and Olin Shivers. Environment analysis via ΔCFA. In Peyton Jones [130], pages 127–140. ISBN 1-59593-027-2.

[118] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. MIT Press, 1997. ISBN 0262631814.

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

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

[121] Gail C. Murphy and Karl J. Lieberherr, editors. *Aspect Oriented Software Development*. ACM Press, March 2004. ISBN 1-58113-842-3.

[122] Chetan R. Murthy. A computational analysis of girard's translation and LC. In *Logic in Computer Science*, pages 90–101. IEEE, June 1992.

[123] Torsten Nelson, Donald D. Cowan, and Paulo S. C. Alencar. Supporting formal verification of crosscutting concerns. In Yonezawa and Matsuoka [170], pages 153–169. ISBN 3-540-42618-3.

[124] Paniti Netinant, Tzilla Elrad, and Mohamed E. Fayad. A layered approach to building open aspect-oriented systems: a framework for the design of on-demand system demodularization. *Communications of the ACM*, 44(10): 83–85, 2001.

[125] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, November 1999. ISBN 3540654100.

[126] Oscar Nierstrasz. Regular types for active objects. In *Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, 1993.

[127] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[128] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns in hyperspace. Technical Report RC 21452(96717)16APR99, IBM, 1999.

[129] Simon Peyton Jones. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, pages 47–96. IOS Press, 2001. ISBN 1 58603 1724. Presented at the 2000 Marktoberdorf Summer School.

[130] Simon Peyton Jones, editor. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2006. ACM Press. ISBN 1-59593-027-2.

[131] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1993.

[132] Benjamin C. Pierce and David N. Turner. Local type inference. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 252–265, 1998.

[133] Christian Queinnec. Continuation conscious compilation. *Lisp Pointers*, 6 (1), 1993.

[134] Christian Queinnec. Locality, causality and continuations. In *Lisp and Functional Programming*, pages 91–102, 1994.

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

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

[137] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[138] Martin Rinard, Alexandru Sălcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In Dwyer and Taylor [59], pages 147–158. ISBN 1-58113-855-5.

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

[140] Amr Sabry. *The Formal Relationship between Direct and Continuation-passing Style Optimizing Compilers: A Synthesis of Two Paradigms*. PhD thesis, Rice University, 1994.

[141] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3–4):289–360, 1993.

[142] Alexandru Sălcianu and Martin C. Rinard. Purity and side effect analysis for java programs. In Cousot [37], pages 199–215. ISBN 3-540-24297-X.

[143] Peter Selinger. Control categories and duality: on the categorical semantics of the $\lambda - \mu$ calculus. *Mathematical Structures of Computer Science*, 11(2): 207–260, 2001.

[144] D. Sereni and Oege de Moor. Static analysis of aspects. In Griswold and Akşit [80], pages 30–39, 2003.

[145] Chung-chieh Shan. Shift to control. In Olin Shivers and Oscar Waddell, editors, *Scheme Workshop*, 1999.

[146] Chung-chieh Shan. From shift and reset to polarized logic. Manuscript, 2003. URL `http://www.eecs.harvard.edu/~ccshan/polar/paper.pdf`.

[147] Olin Shivers. Control flow analysis in scheme. In R. L. Wexelblat, editor, *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174. ACM Press, June 1988. ISBN 0-89791-269-1.

[148] Olin Shivers. Higher-order control-flow analysis in retrospect: Lessons learned, lessons abandoned. In Kathryn S. McKinley, editor, *SIGPLAN Notices*, volume 39, pages 257–269. ACM Press, April 2004. ISBN 1-58113-623-4.

[149] Marcelo Sihman and Shmuel Katz. Superimpositions and aspect-oriented programming. *Computer Journal*, 46(5):529–541, 2003.

[150] Christian Skalka and Francois Pottier. Syntactic type soundness for HM(X). *Electronic Notes in Theoretical Computer Science*, 75, 2003.

[151] Christian Skalka, Scott Smith, and David Van Horn. A type and effect system for flexible abstract interpretation of java (extended abstract). *Electronic Notes in Theoretical Computer Science*, 131:111–124, 2005.

[152] Olaf Spinczyk and Daniel Lohmann. Using AOP to develop architecture-neutral operating system components. In *SIGOPS European Workshop*, pages 188–192. ACM Press, September 2004.

[153] Guy Lewis Steele, Jr. *RABBIT: A Compiler for Scheme*. PhD thesis, MIT, 1978. Also MIT AITR 474.

[154] Guy Lewis Steele, Jr. Building interpreters by composing monads. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 472–492, January 1994.

[155] Christopher Strachey. Fundamental concepts in programming languages. 1967 International Summer School in Computer Programming, Copenhagen, 1967. Reprinted in *Higher-Order and Symbolic Computation*, 13 (1/2):11–49, 2000.

[156] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *ACM Transactions on Software Engineering* 12(1):157–171, 1986.

[157] Greg Sullivan. Aspect-oriented programming with reflection and meta-object protocols. *Communications of the ACM*, 44(10):95–97, 2001.

[158] Martin Sulzmann. *A General Framework for Hindley/Milner Type Systems With Constraints*. PhD thesis, Yale, 2000.

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

[160] Philip Wadler. The essence of functional programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, January 1992.

[161] Philip Wadler. Monads and composable continuations. In *Lisp and Functional Programming*, pages 39–56, January 1994.

[162] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer-Verlag, May 1995. ISBN 3-540-59451-5.

[163] Philip Wadler. The marriage of effects and monads. In *International Conference on Functional Programming*, pages 63–74, 1998.

[164] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *Transactions on Computational Logic*, 4(1):1–32, 2003.

[165] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In Colin Runciman and Olin Shivers, editors, *International Conference on Functional Programming*, August 2003. ISBN 1-58113-756-7.

[166] Mitchell Wand and Daniel P. Friedman. Compiling lambda-expressions using continuations and factorizations. *Computer Languages*, 3(4):241–263, 1978.

[167] Mitchell Wand, Gregor Kiczales, and Christopher J. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In Leavens and Cytron [103], pages 1–8.

[168] Mitchell Wand, Gregor Kiczales, and Christopher J. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In Martin Odersky, editor, *Foundations of Object Oriented Languages*, January 2002.

[169] Mitchell Wand, Gregor Kiczales, and Christopher J. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(4):890–910, September 2004.

[170] Akinori Yonezawa and Satoshi Matsuoka, editors. *Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192, September 2001. Springer-Verlag. ISBN 3-540-42618-3.

Appendices

Appendix A

AspectScheme CEKS Semantics

A.1 Syntactic Categories

Expressions	M	$::=$	V
		$\mid$	x
		$\mid$	$(M\ M)$
		$\mid$	$(o\ M\ \ldots)$
		$\mid$	$(\textbf{if}\ M\ M\ M)$
		$\mid$	$(\textbf{set!}\ x\ M)$
		$\mid$	$(\textbf{around}\ M\ M\ M)$
		$\mid$	$(\textbf{fluid-around}\ M\ M\ M)$
		$\mid$	$(\textbf{app/prim}\ M\ M)$

x :: identifier

Expression closures	MC	$::=$	$\langle M, E, A\rangle$

Values	V	$::=$	$(\lambda\,(x)\ M)_t$
		$\mid$	true
		$\mid$	false
		$\mid$	empty
		$\mid$	(cons $VC\,VC$)

t :: source location tag

Value closures	VC	$::=$	$\langle V, E, A\rangle$

Continuations	K	$::=$	mt-k
		$\mid$	$\langle$app1-k $MC, E, A, K\rangle$
		$\mid$	$\langle$app2-k $VC, E, A, K\rangle$
		$\mid$	$\langle$if-k $MC, MC, K\rangle$
		$\mid$	$\langle$set-k $x, E, A, K\rangle$
		$\mid$	$\langle$around1-k $\mathit{scope}, MC, MC, K\rangle$
		$\mid$	$\langle$around2-k $\mathit{scope}, VC, MC, K\rangle$
		$\mid$	$\langle$markapp-k $VC, K\rangle$
		$\mid$	$\langle$appprim1-k $MC, A, K\rangle$
		$\mid$	$\langle$appprim2-k $VC, A, K\rangle$
		$\mid$	$\langle$op-k $o, \langle VC, \ldots\rangle, \langle MC, \ldots\rangle, K\rangle$

Stores	S	$::$	$\langle\{\ell\}, \ell \rightarrow VC\rangle$
	S_0	$\equiv$	$\langle\{\ell_0\}, \ell \mapsto \text{error}\rangle$

ℓ :: store location

ℓ_0 = fixed store location

Environments $E \quad :: \quad \langle \ell, x \rightarrow \ell \rangle$

$E_0 \quad \equiv \quad \langle \ell_0, x \mapsto \text{error} \rangle$

$$\langle\langle \ell, e\rangle, \langle L, s\rangle\rangle + \{x \mapsto VC\} \equiv \langle\langle \ell_e, e[x \mapsto \ell_v]\rangle, \langle L \cup \{\ell_e\}, s[\ell_v \mapsto VC]\rangle\rangle \text{ where } \ell_e, \ell_v \notin L \cup \text{dom}(S)$$

$$\langle E, S\rangle + \{x_1 \mapsto VC_1, \ldots, x_n \mapsto VC_n\} \equiv \langle E, S\rangle + \{x_1 \mapsto VC_1\} + \cdots + \{x_n \mapsto VC_n\}$$

Advice environments $A \quad :: \quad \{\langle \mathit{scope}, VC, VC\rangle\}$

$A_0 \quad \equiv \quad \emptyset$

$\mathit{scope} \in \{\texttt{static}, \texttt{dynamic}\}$

Primitive operations o ::= *eq?*
| *cons*
| *first*
| *rest*
| *empty?*

A.2 Transition Rules

Initialization and Termination

$$M \ \triangleright_{init} \ \langle\langle M, E_0, A_0\rangle, \mathsf{mt\text{-}k}, S_0\rangle$$

$$\langle\langle V, E, A\rangle, \mathsf{mt\text{-}k}, S\rangle \ \blacktriangleright_{term} \ V$$

Literals

There are no transition rules for literal (value) expressions; they will either appear as the whole program (and hence become value closures by initialization), or they will become closures as their enclosing expression is evaluated.

Variables

$$\langle\langle x, E, A\rangle, K, S\rangle \ \triangleright_{var} \ \langle S(E(x)), K, S\rangle$$

If

$$\langle\langle(\mathbf{if}\ M M_{\text{then}} M_{\text{else}}), E, A\rangle, K, S\rangle$$
$$\triangleright_{if} \ \langle\langle M, E, A\rangle, \langle \mathsf{if\text{-}k}\ \langle M_{\text{then}}, E, A\rangle, \langle M_{\text{else}}, E, A\rangle, K\rangle, S\rangle$$

$$\langle\langle \mathsf{true}, E, A\rangle, \langle \mathsf{if\text{-}k}\ MC_{\text{then}}, MC_{\text{else}}, K\rangle, S\rangle \ \blacktriangleright_{if} \ \langle MC_{\text{then}}, K, S\rangle$$

$$\langle\langle \mathsf{false}, E, A\rangle, \langle \mathsf{if\text{-}k}\ MC_{\text{then}}, MC_{\text{else}}, K\rangle, S\rangle \ \blacktriangleright_{if} \ \langle MC_{\text{else}}, K, S\rangle$$

Continuation Marks

$$\langle VC, \langle \mathsf{markapp\text{-}k}\ VC_{\text{fun}}, K\rangle, S\rangle \ \triangleright_{mark} \ \langle VC, K, S\rangle$$

Set!

$$\langle\langle(\textbf{set!}\ x\ M), E, A\rangle, K, S\rangle \ \rhd_{set}\ \langle\langle M, E, A\rangle, \langle \mathsf{set\text{-}k}\ x, E, A, K\rangle, S\rangle$$

$$\langle VC, \langle \mathsf{set\text{-}k}\ x, E, A, K\rangle, S\rangle \ \blacktriangleright_{set}\ \langle VC, K, S[E(x) \mapsto VC]\rangle$$

Primitive Operations

$$\langle\langle(o\ M_1 \ldots M_n), E, A\rangle, K, S\rangle$$
$$\rhd_{prim}\ \langle\langle M_1, E, A\rangle, \langle \mathsf{op\text{-}k}\ o, \langle\rangle, \langle\langle M_2, E, A\rangle, \ldots, \langle M_n, E, A\rangle\rangle, K\rangle, S\rangle$$

$$\langle VC_m, \langle \mathsf{op\text{-}k}\ o, \langle VC_{m-1}, \ldots, VC_1\rangle, \langle MC_{m+1}, \ldots, MC_n\rangle, K\rangle, S\rangle$$
$$\blacktriangleright_{prim}\ \langle MC_{m+1}, \langle \mathsf{op\text{-}k}\ o, \langle VC_m, \ldots, VC_1\rangle, \langle MC_{m+2}, \ldots, MC_n\rangle, K\rangle, S\rangle$$

$$\langle VC_n, \langle \mathsf{op\text{-}k}\ o, \langle VC_{n-1}, \ldots, VC_1\rangle, \langle\rangle, K\rangle, S\rangle \ \blacktriangleright_{prim}\ \langle \delta(o, VC_1, \ldots, VC_n), K, S\rangle$$

where

$$\delta(\mathit{empty?}, VC) = \begin{cases} \langle \mathsf{true}, E_0, A_0\rangle & \text{if } VC = \langle \mathsf{empty}, E, A\rangle \\ \langle \mathsf{false}, E_0, A_0\rangle & \text{otherwise} \end{cases}$$

$$\delta(\mathit{cons}, VC_1, VC_2) = \langle(\mathsf{cons}\ \ VC_1\, VC_2), E_0, A_0\rangle$$

$$\delta(\mathit{first}, \langle(\mathsf{cons}\ \ VC_1\, VC_2), E, A\rangle) = VC_1$$

$$\delta(\mathit{rest}, \langle(\mathsf{cons}\ \ VC_1\, VC_2), E, A\rangle) = VC_2$$

$$\delta(\mathit{eq?}, \langle(\lambda\,(x)\ \ M)_t, \langle \ell, e\rangle, A\rangle, \langle(\lambda\,(x')\ \ M')_{t'}, \langle \ell', e'\rangle, A'\rangle)$$
$$= \begin{cases} \langle \mathsf{true}, E_0, A_0\rangle & \text{if } t = t' \text{ and } \ell = \ell' \\ \langle \mathsf{false}, E_0, A_0\rangle & \text{otherwise} \end{cases}$$

Around and **Fluid-around**

$\langle\langle(\textbf{around}\ M_{\text{pc}} M_{\text{adv}} M), E, A\rangle, K, S\rangle$
$\quad \triangleright_{\mathit{around}}\ \langle\langle M_{\text{pc}}, E, A\rangle, \langle \textsf{around1-k}\ \texttt{static}, \langle M_{\text{adv}}, E, A\rangle, \langle M, E, A\rangle, K\rangle, S\rangle$

$\langle\langle(\textbf{fluid-around}\ M_{\text{pc}} M_{\text{adv}} M), E, A\rangle, K, S\rangle$
$\quad \triangleright_{\mathit{around}}\ \langle\langle M_{\text{pc}}, E, A\rangle, \langle \textsf{around1-k}\ \texttt{dynamic}, \langle M_{\text{adv}}, E, A\rangle, \langle M, E, A\rangle, K\rangle, S\rangle$

$\langle VC_{\text{pc}}, \langle \textsf{around1-k}\ \mathit{scope}, MC_{\text{adv}}, MC, K\rangle, S\rangle$
$\quad \blacktriangleright_{\mathit{around}}\ \langle MC_{\text{adv}}, \langle \textsf{around2-k}\ \mathit{scope}, VC_{\text{pc}}, MC, K\rangle, S\rangle$

$\langle VC_{\text{adv}}, \langle \textsf{around2-k}\ \mathit{scope}, VC_{\text{pc}}, \langle M, E, A\rangle, K\rangle, S\rangle$
$\quad \blacktriangleright_{\mathit{around}}\ \langle\langle M, E, A \cup \{\langle \mathit{scope}, VC_{\text{pc}}, VC_{\text{adv}}\rangle\}\rangle, K, S\rangle$

App/prim

$\langle\langle(\textbf{app/prim}\ M_{\text{fun}} M_{\text{arg}}), E, A\rangle, K, S\rangle$
$\quad \triangleright_{\mathit{app/prim}}\ \langle\langle M_{\text{fun}}, E, A\rangle, \langle \textsf{appprim1-k}\ \langle M_{\text{arg}}, E, A\rangle, A, K\rangle, S\rangle$

$\langle VC_{\text{fun}}, \langle \textsf{appprim1-k}\ MC_{\text{arg}}, A_{\text{app}}, K\rangle, S\rangle$
$\quad \blacktriangleright_{\mathit{app/prim}}\ \langle MC_{\text{arg}}, \langle \textsf{appprim2-k}\ VC_{\text{fun}}, A_{\text{app}}, K\rangle, S\rangle$

$\langle VC_{\text{arg}}, \langle \textsf{appprim2-k}\ \langle(\lambda\,(x)\ M)_t, E, A_{\text{fun}}\rangle, A_{\text{app}}, K\rangle, S\rangle$
$\quad \blacktriangleright_{\mathit{app/prim}}\ \langle\langle M, E', A'\rangle, K, S'\rangle$
where

$$\begin{aligned}\langle E', S'\rangle &= \langle E, S\rangle + \{x \mapsto VC_{\text{arg}}\}\\ A' &= A_{\text{app}}|_{\texttt{dynamic}} \cup A_{\text{fun}}|_{\texttt{static}}\end{aligned}$$

Function applications

$$\langle\langle(M_{\text{fun}} M_{\text{arg}}), E, A\rangle, K, S\rangle$$
$$\quad \triangleright_{app}\ \langle\langle M_{\text{fun}}, E, A\rangle, \langle \textsf{app1-k}\ \langle M_{\text{arg}}, E, A\rangle, E, A, K\rangle, S\rangle$$

$$\langle VC_{\text{fun}}, \langle \textsf{app1-k}\ MC_{\text{arg}}, E_{\text{app}}, A_{\text{app}}, K\rangle, S\rangle$$
$$\quad \blacktriangleright_{app}\ \langle MC_{\text{arg}}, \langle \textsf{app2-k}\ VC_{\text{fun}}, E_{\text{app}}, A_{\text{app}}, K\rangle, S\rangle$$

$$\langle VC_{\text{arg}}, \langle \textsf{app2-k}\ \langle(\lambda\,(x)\ M)_t, E_{\text{fun}}, A_{\text{fun}}\rangle, E_{\text{app}}, A_{\text{app}}, K\rangle, S\rangle$$
$$\quad \blacktriangleright_{app}\ \langle\langle M', E', A_{\text{app}}\rangle, K', S'\rangle$$

where

$$
\begin{aligned}
M' &= (\textbf{app/prim}\ W[\![\,|\ A_{\text{app}}\ |\,]\!]\ \mathit{arg})\\
K' &= \langle \textsf{markapp-k}\ \langle(\lambda\,(x)\ M)_t, E_{\text{fun}}, A_{\text{fun}}\rangle, K\rangle\\
\langle E', S'\rangle &= \langle E_{\text{app}}, S\rangle\\
&+\ \{\mathit{fun} \mapsto \langle(\lambda\,(x)\ M)_t, E_{\text{fun}}, A_{\text{fun}}\rangle,\ \mathit{arg} \mapsto VC_{\text{arg}},\ \mathit{jp{*}} \mapsto J[\![K']\!]\}\\
&+\ \{\mathit{pc}^i \mapsto VC_{\text{pc}^i}, \mathit{adv}^i \mapsto VC_{\text{adv}^i} \mid \langle \mathit{scope}^i, VC_{\text{pc}^i}, VC_{\text{adv}^i}\rangle \in A_{\text{app}}\}\\
W[\![i]\!] &= \begin{cases} \mathit{fun} & \text{if } i = 0\\ (\textbf{app/prim}\ (\lambda\,(f)\ (\textbf{if}\ (\textbf{app/prim}\ \mathit{pc}^i\ \mathit{jp{*}}) \\ \qquad\qquad (\textbf{app/prim}\ \mathit{adv}^i\ f) \\ \qquad\qquad f)) \\ \qquad W[\![i-1]\!]) & \text{otherwise} \end{cases}\\
J[\![K]\!] &= \begin{cases} \langle \textsf{empty}, E_0, A_0\rangle & \text{if } K = \textsf{mt-k}\\ \langle(\textsf{cons}\ VC\, J[\![K']\!]), E_0, A_0\rangle & \text{if } K = \langle \textsf{markapp-k}\ VC, K'\rangle\\ J[\![K']\!] & \text{if } K = \langle \ldots, K'\rangle \end{cases}
\end{aligned}
$$

Appendix B

AspectScheme 2.3 Implementation

This appendix contains the implementation for AspectScheme 2.3, available from the PLaneT archive via the following PLT Scheme preamble:

```
(require (planet "aspect-scheme2.ss" ("cdutchyn" "aspect-scheme.plt" 2 1)))
```

at the start of your program.

The source code is also available for download at `http://www.cs.ubc.ca/~cdutchyn/downloads/AspectScheme/aspect-scheme2.plt`.

```
;;;
;;; AspectScheme v. 2.3 – with bindings, execution join points, and top-level aspects.
;;; Copyright (c) 2005, 2006 by Christopher Dutchyn (cdutchyn@cs.ubc.ca);
;;; all rights reserved.
;;;

(module aspect-scheme2 mzscheme
  (require (only (lib "list.ss") foldl foldr))

;; Join Point
;;                proc          args
;; jp ::= call-jp a->b   a  ;; procedure application ('a' can be values (ie. tuple ...)
;;     |  exec-jp a->b   a  ;; procedure execution (cannot be advised only matched)
;;     |  adv-jp  adv    c  ;; advice execution ... 'c' can be values as well

;; Pointcut
;; pc :: [jp]*jp*[jp]->c ;; above * jp * below
```

```
;; Advice
;; adv :: (a->b)->c->(a->b)

;; Aspect
;; aspect ::=    fluid-around pc adv body  ;; dynamic scoping
;;         |           around pc adv body  ;; lexical scoping
;;         |  toplevel-around pc adv       ;; top-level scoping (i.e. body is rest of repl)
;;
;; Other kinds of advice (before, after) are special cases; using them might inform a type-
;; checker and enable it to recognize behaviour as extensional rather than superpositional.
;;
;; (before pc                | (around pc
;;                           |     (λ (proceed)
;;     (λ ctxt               |       (λ ctxt
;;       (λ arg              |         (λ args
;;         ...adv-body...))  |           ...adv-body...
;;                           |           (proceed args))))
;;   body)                   |   body)

;; (after pc                 | (around pc
;;                           |     (λ (proceed)
;;     (λ ctxt               |       (λ ctxt
;;          (λ args          |         (λ args
;;                           |           (let-values
;;                           |               ([r (with-handlers ([(λ (x) #t)
;;                           |                                    (λ (x) ...adv-body...
;;                           |                                           raise x)])
;;                           |                     (proceed args)])
;;         ...adv-body...))  |             ...adv-body...
;;                           |             (values r)))))
;;   body)                   |   body)

;; (after-throwing pc        | (around pc
;;                           |     (λ (proceed)
;;     (λ ctxt               |       (λ ctxt
;;       (λ args             |         (λ args
;;                           |           (with-handlers ([(λ (x) #t)
;;         ...adv-body...))  |                            (λ (x) ...adv-body...
;;                           |                                   raise x)])
;;                           |             (proceed args)))))
;;   body)                   |   body)
;;
```

```
;; (after-returning pc        | (around pc
;;                            |     (λ (proceed)
;;     (λ ctxt                |       (λ ctxt
;;       (λ args              |         (λ args
;;                            |           (let-values ([r (proceed args)])
;;         ...adv-body...))   |             ...adv-body...
;;                            |             (values r)))))
;;   body)                    |   body)
```

```
(define-syntax fluid-let-parameter
  (syntax-rules ()
    [(_ ([p v]) e ...)
     (let ([y v])
       (let ([swap (λ () (let ([t (p)])
                           (p y)
                           (set! y t)))])
         (dynamic-wind swap (λ () e ...) swap)))]))

;; aspect structure
(define-struct aspect (pc adv))

(define-struct jp (target args))
(define-struct (call-jp jp)())
(define-struct (exec-jp jp)())
(define-struct (adv-jp  jp)())

;; join points implemented via continuation marks
(define (jp-context)
  (continuation-mark-set→list
   (current-continuation-marks)
   'joinpoint))

(define-syntax with-joinpoint
  (syntax-rules ()
    [(_ jp body ...)
     ((λ (x) x)
      (with-continuation-mark 'joinpoint jp
        (begin body ...)))]))
```

```
;; dynamically-scoped aspects
(define dynamic-aspects (make-parameter '()))
(define  static-aspects (make-parameter '()))

(define-syntaxes (fluid-around around)
  (let ([round (λ (param)
                 (λ (stx)
                   (syntax-case stx ()
                     [(_ pc adv body0 ...)
                      (quasisyntax/loc stx
                        (fluid-let-parameter ([#,param (cons (make-aspect pc adv)
                                                             (#,param))])
                          body0 ...))])))])
    (values (round #'dynamic-aspects)   ;dynamically-scoped
            (round #'static-aspects)))) ;lexically-scoped

;; lexically-scoped aspects
(define-syntax lambda/static
  (syntax-rules ()
    [(_ params body ...)
     (let ([aspects (static-aspects)])
       (λ params
         (fluid-let-parameter ([static-aspects aspects])
           body ...)))]))

;; top-level aspects
(define toplevel-aspects (make-parameter '()))

(define (toplevel-around pc adv)
  (toplevel-aspects (cons (make-aspect pc adv) (toplevel-aspects))))

;; current aspects – in decending order of application!
(define (current-aspects)
  (append (dynamic-aspects)
          (static-aspects)
          (toplevel-aspects)))
```

```
;; weaver
;; replacement for #%app
(define-syntax app/weave
  (syntax-rules ()
    [(_ f a ...) (app/weave/rt f a ...)]))

(define (app/weave/rt fun-val . arg-vals)
  (if (primitive? fun-val)
      (apply fun-val arg-vals)
      (let ([jp (make-call-jp fun-val arg-vals)])
        (with-joinpoint jp
          (apply (weave (λ arg-vals
                           (with-joinpoint (make-exec-jp fun-val arg-vals)
                             (apply fun-val arg-vals)))
                         '() jp (jp-context)
                         (current-aspects))
                 arg-vals)))))

(define (weave fun-val jp- jp jp+ aspects)
  (foldr (λ (aspect fun)
           (cond
             [((aspect-pc aspect) jp- jp jp+)
              ⇒ (λ (ctxt-vals)
                   (with-joinpoint (make-adv-jp (aspect-adv aspect) ctxt-vals)
                     (apply ((aspect-adv aspect) fun) ctxt-vals)))]
             [else fun]))
         fun-val
         aspects))
```

```
;; pointcuts – strict combinators
(define ((&& . pcs) jp- jp jp+)
  (let loop ([pcs  pcs]
             [res  '()])
    (if (null? pcs)
        (reverse res)
        (let ([r ((car pcs) jp- jp jp+)])
          (and r
               (loop (cdr pcs) (append (reverse r) res)))))))

(define ((|| . pcs) jp- jp jp+)
  (let loop ([pcs pcs])
    (and (not (null? pcs))
         (or ((car pcs) jp- jp jp+)
             (loop (cdr pcs))))))

(define ((! pc) jp- jp jp+)
  (and (not (pc jp- jp jp+))
       '()))

;; pointcuts – 'binding'
(define (target jp- jp jp+)
  (list (jp-target jp)))

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

(define ((some-args as) jp- jp jp+)
  (foldl (λ (a v l)
           (if a
               (cons v l)
               l))
         '()
         as
         (jp-args jp)))
```

```
;; pointcuts – structural
(define (top? jp- jp jp+)
  (and (null? jp+)
       '()))

(define (top pc)
  (&& pc
      (! (cflowbelow pc))))

(define ((below pc) jp- jp jp+)
  (and (not (null? jp+))
       (pc (cons jp jp-) (car jp+) (cdr jp+))))

(define ((above pc) jp- jp jp+)
  (and (not (null? jp-))
       (pc (cdr jp-) (car jp-) (cons jp jp+))))

(define (bottom pc)
  (&& pc
      (! (cflowabove pc))))

(define (bottom? jp- jp jp+)
  (and (null? jp-)
       '()))

;; pointcuts - compatibility
(define (cflow pc)
  ((cflow-walk below top?) pc))

(define (within f)
  (cflowbelow (&& (exec f)
                  (! (cflowabove call?)))))
```

```
;; pointcuts – fundamental
(define ((kind= k?) jp- jp jp+)
  (and (k? jp)
       '()))

(define call? (kind= call-jp?))

(define exec? (kind= exec-jp?))

(define adv? (kind= adv-jp?))

(define ((target= f) jp- jp jp+)
  (and (eq? f (jp-target jp))
       '()))

(define (call f)
  (&& call?
      (target= f)))

(define (exec f)
  (&& exec?
      (target= f)))

(define (adv a)
  (&& adv?
      (target= a)))

;; pointcuts - higher-order recursive
(define (((cflow-walk step end) pc) jp- jp jp+)
  ((|| pc
       (&& (! end)
           (step ((cflow-walk step end) pc)))) jp- jp jp+))
```

```
;; pointcuts - higher-order points-free
(define (cflowtop pc)
  (cflowbelow (top pc)))

(define (cflowbelow pc)
  (below ((cflow-walk below top?) pc)))

(define (cflowabove pc)
  (above ((cflow-walk above bottom) pc)))

(define (cflowbottom pc)
  (cflowbelow (bottom pc)))

;; language definition
(provide (all-from-except mzscheme #%app λ)
         (rename app/weave #%app)
         (rename #%app app/prim)
         (rename lambda/static λ)

          fluid-around
          around
          toplevel-around

          && || !
          top? top below above bottom bottom?
          target args some-args
          call? exec? adv? call exec adv
          cflowtop cflowbelow cflowbottom cflowabove
          cflow within
          ))
```

Appendix C

PROC Implementation

This appendix contains the implementation for the Proc language, as described in Chapter 2, and extended with exceptions and threads.

C.1 Syntax

```
;;;
;;; Syntax – mutually-recursive, first-order procedural [WKD04]
;;;
;;program
(define-struct pgm [decls body]) ; PGM ::= (id * decl)... * exp

;; declarations
(define-struct procD [ids body])  ; DECL ::= PROC id... * exp
(define-struct globD [])          ; — GLOBAL

;; expressions
(define-struct litX [val])              ; EXP ::= 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...
```

C.2 Parser

```
(define (parse-pgm s) ;: sexp → pgm
  (make-pgm (map parse-named-decl (car s))
            (parse-exp (cadr s))))

(define (parse-named-decl i+s) ;: sexp → (id * decl)
  (cons (car i+s)
        (parse-decl (cdr i+s))))

(define (parse-decl s) ;: sexp → decl
  (case (car s)
    [(proc) (make-procD (cadr s) (parse-exp (caddr s)))]
    [(global) (make-globD)]
    [(advise) (make-advD (parse-pc (cadr s)) (parse-exp (caddr s)))]
    [else (error 'parse-decl "not a decl: ~a" s)]))

(define (parse-exp s) ;: sexp → exp
  (cond [(number? s) (make-litX s)]
        [(boolean? s) (make-litX s)]
        [(symbol? s) (make-varX s)]
        [(pair? s) (case (car s)
                     [(if) (make-ifX (parse-exp (cadr s))
                                     (parse-exp (caddr s))
                                     (parse-exp (cadddr s)))]
                     [(seq) (make-seqX (map parse-exp (cdr s)))]
                     [(let) (make-letX (map car (cadr s))
                                       (map parse-exp (map cadr (cadr s)))
                                       (parse-exp (caddr s)))]
                     [(get) (make-getX (cadr s))]
                     [(set) (make-setX (cadr s) (parse-exp (caddr s)))]
                     [(call) (make-appX (cadr s)
                                        (map parse-exp (cddr s)))]
                     [(proceed) (make-pcdX (map parse-exp (cdr s)))]
                     [else (error 'parse "not an exp: ~a" s)])]
        [else (error 'parse "not an exp: ~a" s)]))
```

```
(define (parse-pc s)
  (case (car s)
    [(get) (make-getC (cadr s))]
    [(set) (make-setC (cadr s) (caddr s))]
    [(call) (make-callC (cadr s) (cddr s))]
    [(exec) (make-execC (cadr s) (cddr s))]
    [(or) (make-orC (map parse-pc (cdr s)))]
    [(not) (make-notC (parse-pc (cdr s)))]
    [else (error 'parse-pc "not a pointcut ~a" s)]))
```

C.3 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))... → !
  (set! *globs* '())
  (set! *procs* prims)
  (set! *advs* '())
  (for-each (λ (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*)))
```

C.4 Evaluator

```
;;; Evaluator

;;; frames
;; auxiliary
(define-struct testF [then else env]) ; FRM ::= TEST exp exp env :: !bool
(define-struct bindF [ids body env])  ; — BIND id... exp env :: !val...
(define-struct nextF [exps env])      ; — NEXT 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

(define-struct pcdF [v→v+f advs])     ; — PCD val... → val+frm adv... :: !val...

;;; continuations ::= 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))))
```

```
;;; evaluator – expression side
(define (eval x r k) ;: (exp * env * cont) → !
  ;(display `(E ,x ,k))(newline)
  (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))]
        [(seqX? x) (let ([x* (seqX-exps x)])
                     (if (null? x*)
                         (apply k 0)
                         (evseq (car x*) (cdr x*) r k)))]
        [(letX? x) (evlis (letX-rands x)
                          r
                          (push (make-bindF (letX-ids x) (letX-body 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))]
        [(pcdX? x) (evlis (pcdX-rands x)
                          r
                          (push (make-pcdF (lookup-env r '%proceed)
                                           (lookup-env r '%advs))
                                k))]
        [else (error 'eval "not an exp: ~a" x)]))
```

```
(define (evseq x x* r k) ;: (val * exp... * env * cont) → !
  (eval x
        r
        (if (null? x*)
            k
            (push (make-nextF x* r)
                  k))))

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

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

(define (apply k v) ;: (cont * val) → !
  ;(display `(A ,k ,v))(newline)
  (((pop halt
         step)
    k)
   v))
```

```
;;; 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)]
        [(nextF? f) (let ([x* (nextF-exps f)])
                      (evseq (car x*) (cdr x*) (nextF-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)))]
        [(bindF? f) (eval (bindF-body f)
                          (extend-env (bindF-ids f)
                                      v
                                      (bindF-env f))
                          k)]
```

```
      ;; non-auxiliary frames
      [(getF? f) (apply k
                        (get-glob v))]
      [(rhsF? f) (apply (push (make-setF v)
                              k)
                        (lookup-glob (rhsF-id f)))]
      [(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)])]
      [(pcdF? f) (let-values ([(v1 f1) ((pcdF-v→v+f f) v)])
                   (((adv-step (pcdF-advs f)) f1 k) v1))]
      [else (error 'step "not a frame: ~a" f)]))

(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) ⇒ (λ (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))])))
```

```
;;; defined once the *adv* are elaborated
(define step ;(adv-step *adv*)) ;: (frm * cont) → val → !
  #f)
```

C.5 AOP Constructs

```
;;; pointcuts and advice aop

;; pointcuts
;; effective continuation frame matching
(define-struct getC [gid])       ; PCUT ::= 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

(define-struct andC [pcs]) ; — ANDPC pcut...
;; declarations
(define-struct advD [pc body]) ; DECL +:= ADVICE pcut exp

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

(define (merge-match m1 m2)
  (make-match
   (append (match-ids m1) (match-ids m2))
   (append (match-vals m1) (match-vals m2))
   (λ (nv)
     (let-values ([(nv1 f1) (match-prcd m1 nv)])
       (match-prcd m2 nv1)))))
```

```
;;; matching
(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 '()
                                   '()
                                   (λ (nv)
                                     (values v f))))]
;; [(andC? c) (let loop ([pcs (andC-pcs c)])
;; (if (null? pcs)
;; (make-match '()
;; '()
;; (lambda (nv)
;; (values v f)))
;; (merge (match-pc (car pcs) v f)
;; (loop (cdr pcs)))))]
```

```
 ;; fundamental pointcuts
[(getC? c) (and (getF? f)
                (eq? (lookup-glob (getC-gid c)) v)
                (make-match '()
                            '()
                            (λ (nv)
                              (values v f))))]
[(setC? c) (and (setF? f)
                (eq? (lookup-glob (setC-gid c)) v)
                (make-match `(,(setC-id c))
                            `(,(setF-val f))
                            (λ (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
                             (λ (nv)
                               (values nv f))))]
[(execC? c) (and (execF? f)
                 (eq? (lookup-proc (execC-pid c)) v)
                 (make-match (execC-ids c)
                             (execF-args f)
                             (λ (nv)
                               (values v (make-execF nv)))))]
[else (error 'match-pc "not a pointcut: ~a" c)]))
```

C.6 Environments

```
;;;
;;; Environments
;;;

;;; environment – env :: id → val

(define (empty-env i) ;: env
  (error 'lookup "not found: ~a" i))

(define ((extend-env i* v* r) i) ;: (id... * val... * env) → id → val
  (let loop ([i* i*] [v* v*])
    (cond [(null? i*) (r i)]
          [(eq? (car i*) i) (car v*)]
          [else (loop (cdr i*) (cdr v*))])))

(define (lookup-env r i)
  (r i))
```

C.7 Top Level

```
;;; top level

(load "env.scm")
(load "syntax.scm")
(load "elab.scm")
(load "eval.scm")
(load "aop.scm")
(load "parse.scm")

(define _prims_
  `([+ ,(lift (λ (vs)
                (+ (car vs)
                   (cadr vs))))]
    [= ,(lift (λ (vs)
                (= (car vs)
                   (cadr vs))))]
    [display ,(lift (λ (vs)
                      (display (car vs))
                      0))]
    [newline , (lift (λ (vs)
                       (newline)
                       0))]
    [abort ,(λ (vs k) ; Felleisen 𝒜 operator
              (apply '() (car vs)))]))

(define (run s)
  (let ([g (parse-pgm s)])
    (elab! _prims_ (pgm-decls g))
    (eval (pgm-body g)
          empty-env
          '())))

(load "tests.scm")
```

Subject Index

$\mathcal{A}$ .... 12.
advice .... 14, 18–24.
  after .... 18.
  around .... 18, 31.
  before .... 18.
  fluid-around .... 31.
  matching .... 20.
advice environment .... 32.
aspect
  assistant .... 93.
  harmless .... 94.
  invasive .... 94.
    weakly .... 94.
  regulative .... 93.
  spectative .... 93.
    weakly .... 93.
  spectator .... 92.
AspectJ .... 24.
AspectScheme .... 25.

behaviour analysis .... 53.

CEKS machine .... 30–31.
closure .... 10.
continuation .... 9, 10.
continuation mark ...... *see* mark, continuation.
cps .... 10.

declare precedence .... 87.
Demeter, Law of .... 63.
dominates .... 87.

effect .... 54.
  concurrency .... 55.
  exception .... 54.
  io .... 55.
  nondeterminism .... 55.
  region .... 59.
  state .... 54.
effect report
  advice .... 65.
  pointcut .... 62.
effect string .... 56.
  advice .... 64.
  join point .... 59.
  pointcut .... 61.
  procedure .... 56.

frame
  auxiliary .... 11.
full abstraction .... 99.

interaction
  compound .... 75, 86–87.
  concurrency .... 85–86.
    asynchronous .... 85.
    mixed .... 85.
    synchronous .... 85.
  control .... 76–78.
    coupling .... 77.
    extension .... 77.
    narrowing .... 77.
    repetition .... 77.
    replacement .... 77.

- data ..... 78–82.
  - actuation ..... 80.
  - independent ..... 79.
  - influence ..... 80.
  - interference ..... 81.
  - observation ..... 80.
  - orthogonal ..... 79.
- exception ..... 84–85.
  - injection ..... 84.
  - masking ..... 84.
- io ..... 82–84.
  - advice only ..... 83.
  - both ..... 83.
  - join point only ..... 83.
  - neither ..... 82.
- simple ..... 75, 76.

join point ..... 14.
- advice execution ..... 86.
- dynamic ..... 15–16.
- principle ..... 15.
- shadow ..... 59.

let
- monadic ..... 10.

mark
- continuation ..... 25, 37.

monad ..... 27.

pointcut ..... 14, 16–18.
- call ..... 17.
- cflow ..... 25, 38.
  - optimization ..... 40.
- exec ..... 17.
- get ..... 17.
- not ..... 17.
- or ..... 17.
- principle ..... 17.
- set ..... 17.

procedure string ..... 58.
proceed ..... 18, 21.

reflection,monadic ..... 102.

reset ..... 102.

scoping
- dynamic ..... 35.
- static ..... 35.

semantics
- big-step ..... 8.
- continuation ..... 9.
- direct ..... 8, 24.
- game ..... 100.
- small-step ..... 10.

shift ..... 102.

Citation Index

Abramsky and Jagadeesan [1994] 100, 103.
Abramsky et al. [2000] ... 100, 103.
Abramsky [2006] .... 100, 103.
Ager et al. [2003] .... 26, 103.
Ager et al. [2005] ..... 10, 25, 103.
Allen et al. [1990] .... 7, 103.
Amtoft et al. [1999] ...... 102, 103.
Andrews [2001] .... 27, 103.
Angus [1997] .... 102, 103.
Appel [1992] .... 56, 103.
Avgustinov et al. [2005] ... 41, 104.
Aßmann and Ludwig [1999] .... 16, 104.
Bergmans and Akşit [2001] ...... 2, 100, 104.
Biagioni et al. [1998] ...... 68, 104.
Biernacka et al. [2005] ... 102, 104.
Biernacki et al. [2006] .... 102, 104.
Blanchet [2003] .... 71, 104.
Bockisch et al. [2004] ..... 48, 104.
Brooks et al. [1982] .... 71, 104.
Bruns et al. [2004] .... 26, 99, 104.
Cardelli [1988] .... 100, 104.
Cardelli [2003] ...... 105, 109, 112.
Clement et al. [2005] ..... 100, 105.
Clements and Felleisen [2003] .. 41, 105.
Clements and Felleisen [2004] .. 41, 105.
Clements and Northrop [2001] .. 2, 105.
Clifton and Leavens [2002] ..... 89, 92, 105.
Clifton et al. [2000] ...... 100, 105.
Clinger [1998] .... 41, 105.
Coady et al. [2004] .... 1, 105.
Cousot and Cousot [1977] ...... 72, 105.
Cousot and Cousot [1993] ...... 72, 105.
Cousot and Cousot [2000] ...... 72, 106.
Cousot and Cousot [2002] ...... 72, 106.
Cousot and Cousot [2004] ...... 72, 106.
Cousot [2005] .... 106, 111, 114.
Damas and Milner [1982] ..... 101, 106.
Damian and Danvy [2003] ..... 99, 106.
Dantas and Walker [2006] ...92, 94, 106.
Dantas et al. [2005] ... 26, 54, 101, 106.
Dantas et al. [to appear] ... 26, 54, 99, 101, 106.
Danvy and Filinski [1990] ...... 72, 107.

Danvy and Hatcliff [1992] . . . . . . 11, 107.
Danvy and Hatcliff [1993] . . . . . . 10, 107.
Danvy and Lawall [] . . . . . 102, 107.
Danvy and Nielson [2003] . . . . . . 12, 107.
Danvy and Pierce [2005] . . . . . . 106, 107, 112.
Danvy et al. [1999] . . . . 99, 106.
Danvy [1994] . . . . 102, 106.
Danvy [2000] . . . . 10, 107.
De Win et al. [2004] . . . . 1, 107.
Douence et al. [2001] . . . . . . 27, 107.
Dutchyn et al. [2002] . . . 8, 24, 107.
Dutchyn et al. [2006] . . . xiv, 25, 29, 37, 47, 99, 108.
Dutchyn [2005] . . . . 29, 107.
Dutchyn [2006] . . . . 25, 29, 107.
Dwyer and Taylor [2004] . . . . . . 108, 111, 114.
Espinosa [1994] . . . . . . 72, 102, 108.
Felleisen and Hieb [1992] . . . 30, 108.
Felleisen et al. [1988] . . . . . . 12, 108.
Felleisen [1988] . . . . 12, 108.
Field et al. [2005] . . . . 102, 108.
Filinski [1989] . . . . 19, 102, 108.
Filinski [1994] . . . . 72, 108.
Filinski [1996] . . . . 72, 108.
Filinski [1999] . . . . 72, 108.
Filman and Friedman [2004] . . . 26, 109.
Filman et al. [2004] . . . . . . 105, 107, 109.
Filman [2001] . . . . 8, 108.
Findler et al. [2002] . . . . 3, 109.
Flanagan et al. [1993] . . . 11, 12, 14, 71, 109.
Friedman et al. [2001] . . . . . . 8, 109.
Führmann [2002] . . . . 102, 109.
Gapeyev and Pierce [2003] . . . . 102, 109.
George et al. [1995] . . . . 71, 109.
Gosling et al. [2000] . . . . 37, 109.
Gray et al. [2001] . . . . 2, 109.
Griswold and Akşit [2003] . . . . . 109, 110, 114.
Harrison [1989] . . . . 56, 58, 72, 109.
Hatcliff and Danvy [1994] . . . . . . 10, 109.
Hilsdale and Hugunin [2004] . . . 41, 110.
Hudak et al. [1992] . . . . 72, 110.
Jones and Duponcheel [1993] . . . 72, 102, 110.
Jouvelot and Gifford [1989] . . . . 15, 19, 54, 69, 72, 110.
Kameyama [2000] . . . . 102, 110.
Katz and Gil [1999] . . . 92, 93, 110.
Katz [2004] . . . . 93, 110.
Katz [2006] . . . . 92, 93, 110.
Kelsey et al. [1998] . . . 8, 34, 38, 41, 110.
Kiczales et al. [1997] . . . 2, 14, 110.
Kiczales et al. [2001] . . . 2, 24, 110.
Kiczales [2003] . . . . xii, 110.
King and Wadler [1993] . . . 72, 102, 111.
Kirli [2002] . . . . 102, 111.
Krishnamurthi et al. [2004] . . . . . 95, 111.
Laddad [2003] . . . . . 88, 90, 91, 111.
Lam et al. [2003] . . . . 72, 111.
Lam et al. [2004] . . . . 72, 111.
Lam et al. [2005] . . . . 102, 111.
Landin [1965] . . . . 9, 111.
Leavens and Cytron [2002] . . . . 111, 112, 116.
Levy [1999] . . . . 53, 111.
Liang et al. [1995] . . . . 102, 111.
Liang [1997] . . . . 72, 111.
Lieberherr et al. [2001] . . . . . . 2, 63, 100, 112.
Lieberherr et al. [2004] . . . . 63, 112.
Lohmann and Spinczyk [2003] . . . 1, 112.

Loucassen and Gifford [1988] . . . 72, 112.
Loucassen [1987] . . . . 72, 112.
Masuhara and Kiczales [2003] . . . 100, 112.
Masuhara et al. [2002] . . . . . . xiv, 8, 112.
Masuhara et al. [2003] . . xiv, 8, 46, 112.
Masuhara et al. [2005] . . . . . 48, 99, 112.
Meyer and Riecke [1988] . . . 72, 112.
Might and Shivers [2006] . . . 72, 112.
Milner et al. [1997] . . . . 48, 112.
Moggi [1989] . . . . 10, 72, 113.
Moggi [1991] . . . . 10, 54, 72, 113.
Murphy and Lieberherr [2004] . . . 104, 110, 113.
Murthy [1992] . . . . 15, 113.
Nelson et al. [2001] . . . . 87, 113.
Netinant et al. [2001] . . . . . . 2, 113.
Nielson et al. [1999] . . . . 53, 71, 72, 113.
Nierstrasz [1993] . . . . 102, 113.
Odersky et al. [1999] . . . . . 101, 113.
Ossher and Tarr [1999] . . . 100, 113.
Peyton Jones [2006] . . . . . . 106, 112, 113.
Peyton Jones and Wadler [1993] . . . 72, 113.
Peyton Jones [2001] . . . . 72, 113.
Pierce and Turner [1998] . . . . . . 101, 113.
Queinnec [1993] . . . . 71, 114.
Queinnec [1994] . . . . 71, 114.
Reynolds [1972] . . . . 8, 11, 114.
Reynolds [1993] . . . . 9, 114.
Reynolds [1998] . . . . 8, 11, 114.
Rinard et al. [2004] . . . . 76, 114.
Roychoudhury and Gray [2005] . . . 16, 114.
Sabry and Felleisen [1993] . . . . . . 54, 114.
Sabry [1994] . . . . 14, 71, 114.
Selinger [2001] . . . . 54, 114.
Sereni and de Moor [2003] . . . . . 41, 114.
Shan [1999] . . . . 102, 114.
Shan [2003] . . . . 102, 114.
Shivers [1988] . . . . . . 56, 70, 71, 115.
Shivers [2004] . . . . 70, 99, 115.
Sihman and Katz [2003] . . . 92, 115.
Skalka and Pottier [2003] . . . . . . 101, 115.
Skalka et al. [2005] . . . . 72, 115.
Spinczyk and Lohmann [2004] . . . 1, 115.
Steele [1978] . . . . 71, 115.
Steele [1994] . . . . 72, 115.
Strachey [1967] . . . . 9, 115.
Strom and Yemini [1986] . . . . . . 102, 115.
Sullivan [2001] . . . . 2, 115.
Sulzmann [2000] . . . . 101, 115.
Sălcianu and Rinard [2005] . . . . . 71, 114.
Thielecke [1997] . . . . 11, 15, 115.
Wadler and Thiemann [2003] . . . 54, 116.
Wadler [1992] . . . . 55, 116.
Wadler [1994] . . . . 72, 102, 116.
Wadler [1995] . . . . 72, 116.
Wadler [1998] . . . . 54, 116.
Walker et al. [2003] . . . 48, 54, 101, 116.
Wand and Friedman [1978] . . . . . 71, 116.
Wand et al. [2002] . . . xiv, 8, 24, 116.
Wand et al. [2004] . . . xiv, 8, 16, 24, 25, 99, 100, 116.
Yonezawa and Matsuoka [2001] . . . 103, 113, 116.

Colophon

This dissertation was typeset using the the $\TeX$ typesetting system created by Donald Knuth, the $\LaTeX\,2_\varepsilon$ document preparation macros created by Leslie Lamport, the *memoir* class designed by Peter Wilson, and the *ubc-diss* package developed by Christopher Dutchyn. Program code is composed using *SLaTeX* developed by Dorai Sitaram. The body text is set 10/15pt $\times$ 32pc with Computer Modern Roman designed by Donald Knuth. Other fonts include Sans, Smallcaps, *Italic, Slanted,* and `Typewriter,` all from Knuth's Computer Modern family.