Parameterized Aspect Calculus: A Core Calculus for the Direct Study of Aspect-Oriented Languages

Curtis Clifton, Gary T. Leavens, and Mitchell Wand

TR #03-13
November 2003

Keywords: Parameterized aspect calculus, object calculus, join point model, point cut description language, aspect-oriented programming, AspectJ, advice, HyperJ, hyperslices, DemeterJ, adaptive methods
2003 CR Categories: D.3.1 [*Programming Languages*] Formal Definitions and Theory — Semantics D.3.2 [*Programming Languages*] Language Classifications — object-oriented languages D.3.3 [*Programming Languages*] Language Constructs and Features — classes and objects

Submitted for publication.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Parameterized Aspect Calculus: A Core Calculus for the Direct Study of Aspect-Oriented Languages

Curtis Clifton
Dept. of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA 50011-1040 USA
cclifton@cs.iastate.edu

Gary T. Leavens
Dept. of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA 50011-1040 USA
leavens@cs.iastate.edu

Mitchell Wand
College of Computer and Information Science
Northeastern University
Boston, MA 02115 USA
wand@ccs.neu.edu

ABSTRACT

Formal study of aspect-oriented languages is difficult because current theoretical models provide a range of features that is too limited and rely on encodings using lower-level abstractions, which involve a cumbersome level of indirection. We present a calculus, based on Abadi and Cardelli's object calculus, that explicitly models a base language and a variety of point cut description languages. This explicit modeling makes clear the aspect-oriented features of the calculus by removing the indirection of some existing models. We demonstrate the generality of our calculus by presenting models for AspectJ's open classes and advice, and HyperJ's compositions, and sketching a model for DemeterJ's adaptive methods.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; D.3.2 [**Programming Languages**]: Language Classifications—*object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects*

Keywords

Parameterized aspect calculus, object calculus, join point model, point cut description language, aspect-oriented programming, AspectJ, advice, HyperJ, hyperslices, DemeterJ, adaptive methods

1. INTRODUCTION

Formal models are helpful in understanding new programming paradigms and exploring their properties. Formal study of aspect-oriented languages is difficult because current theoretical models provide a range of features that is too limited. Most existing formal models focus on the dynamic advice binding, or weaving, features of AspectJ-like languages [8, 17, 19]. These models neglect other aspect-oriented features like AspectJ's open classes [3, 9], HyperJ's compositions [14], and DemeterJ's adaptive methods [11].

Another weakness of current formal models of aspect-oriented languages, is that they rely on encodings using lower-level abstractions. This is similar to the situation early in the formal study of object-oriented languages. At that time, there were studies that modeled object-oriented languages using the lambda calculus. However, this indirect modeling was cumbersome because of the multiple layers [1, §8.6], hampering work on semantics and type theory for the object-oriented paradigm. Abadi and Cardelli developed the object calculus, which directly models object-oriented constructs, to alleviate this problem.

In this paper we introduce the *parameterized aspect calculus*, which is a step towards solving these problems.

Our calculus is based on Abadi and Cardelli's functional object calculus, ς [1, pp. 57–78]. We extend the ς calculus to create a core calculus, $\varsigma_{asp}(\boldsymbol{M})$, where $\boldsymbol{M}$ is a parameter defining one of a variety of point cut description languages. This is typical of design for aspect-oriented languages, where an advice mechanism is added on to an existing non-aspect-oriented base language [3, 16]. In $\varsigma_{asp}(\boldsymbol{M})$, the base programs are written in a variant of the ς calculus, and one can also write advice containing code related to various cross-cutting concerns.

As usual, advice can be triggered at certain points in the execution of the base program. At every reduction step in a $\varsigma_{asp}(\boldsymbol{M})$ program, the semantics consults the given point cut description language, $\boldsymbol{M}$, to find any applicable advice, making each reduction step a potential join point. As with Abadi and Cardelli's direct modeling of objects, this direct modeling of join points is intended to simplify reasoning about them. Because join points are explicit and pervasive in the calculus, theorists can instantiate $\varsigma_{asp}(\boldsymbol{M})$ with different point cut description languages to investigate a large variety of aspect-oriented features. We demonstrate this by presenting models for AspectJ's open classes and advice, and HyperJ's compositions, and sketching a model for DemeterJ's adaptive methods.

2. ADDING ASPECTS TO OBJECTS

In this section we present the syntax and operational semantics of our parmeterized object calculus.

2.1 Parameterized Aspect Calculus

We begin by outlining how the calculus incorporates aspect-oriented features. The key to this is how the operational semantics allows advice code to be executed at most reduction steps. That is, most reduction steps are potential join points. The semantics represents reduction steps abstractly as four-tuples, $\langle \rho, \mathcal{K}, S, k \rangle$, where:

- ρ is one of $\{\textsc{Val}, \textsc{Ivk}, \textsc{Upd}\}$, indicating a value creation, invocation, or update operation,
- $\mathcal{K}$ is a string representing the context in which the subterm to be reduced appears,
- S is the signature, either a basic constant or a set of method labels, of the value created or against which invocation or update is to be performed, and
- k is the *message*, either the label or functional constant, that is to be invoked or updated, or the empty string, ϵ, for value creation.

We refer to such a four tuple as a *join point*, and to the set of all possible join points as the *join point model*, written $\boldsymbol{J}$.

An instance, $\varsigma_{asp}(\boldsymbol{M})$, of the calculus is created by defining a *point cut description language*, $\boldsymbol{M} = \langle \boldsymbol{\mathcal{C}}, match \rangle$. The grammar $\boldsymbol{\mathcal{C}}$ gives the syntax for point cut descriptions; *match* gives their semantics. A piece of advice, $\mathcal{A}$, consists of a *point cut description*, *pcd*, defining a set of join points in the reduction of a term, and a *naked method*, $\varsigma(\overrightarrow{y})b$, so called because they have the form of methods in the ς calculus but are not "clothed" in objects. The function *match* takes a piece of advice and a join point and returns a sequence of naked methods to be executed in place of the original operation. The naked methods may also proceed to the original operation, as discussed below.

The *match* function returns an empty sequence if the point cut description of the given advice does not match the given join point. On the other hand, if the point cut description and join point do match, then the function would typically return a singleton sequence containing the naked method of the advice. (Some point cut description languages that we study return longer sequences, for example, by quantifying over some component of the naked method.)

2.1.1 Notation and Syntax

Before describing the syntax and semantics in detail, it is useful to introduce some notation for sets and sequences. We use an over-line notation, $\overline{l_i}^{\,i \in I}$, to denote a set $\{l_i | i \in I\}$ indexed by the set I. We use a vector notation, $\overrightarrow{b_i}^{\,i \in I}$, to denote a sequence $\langle b_{i_1}, b_{i_2}, \ldots, b_{i_n} \rangle$ indexed by the ordered set $I = \{i_1 < i_2 < \ldots < i_n\}$. For empty I, the notations denote the empty set, also written "$\varnothing$", and the empty sequence, also written "$\bullet$". We omit the index from the notation if it is clear from context. We use a comma as shorthand for set union and we use "+" to denote sequence concatenation.

Figure 1 gives the syntax and meta-syntax for $\varsigma_{asp}(\boldsymbol{M})$. We use x to denote a variable drawn from a countable set *Vars* of variables. Basic constants are denoted by d, drawn from a countable set *Consts*, and functional constants by f, drawn from a countable set *FConsts*. The meta-variable l ranges over a countable set *Labels* of method labels. We assume that *FConsts* and *Labels* are disjoint. The meta-variable S ranges over the set of signatures. The *signature* of an object is its set of labels; a basic constant is its own signature.

Syntax:

$x \in \mathit{Vars} \qquad d \in \mathit{Consts} \qquad f \in \mathit{FConsts} \qquad l \in \mathit{Labels}$

$S \in \boldsymbol{P}(\mathit{Labels}) \cup \mathit{Consts} \qquad \mathit{pcd} \in \boldsymbol{\mathcal{C}}$

programs	$\mathcal{P}$	::=	$a \otimes \overrightarrow{\mathcal{A}}$
terms	a, b, c	::=	$x \mid v \mid a.k \mid$
			$a.l \Leftarrow \varsigma(x)b \mid$
			$\mathsf{proceed}_{\textsc{Val}}() \mid$
			$\mathsf{proceed}_{\textsc{Ivk}}(a) \mid$
			$\mathsf{proceed}_{\textsc{Upd}}(a, \varsigma(x)b) \mid \pi$
values	v	::=	$d \mid [\overline{l_i = \varsigma(x_i)b_i}^{\,i \in I}]$
selectors	k	::=	$l \mid f$
proceed closures	π	::=	$\Pi_{\textsc{Val}}\{\!\{B, v\}\!\}() \mid$
			$\Pi_{\textsc{Ivk}}\{\!\{B, S, k\}\!\}(a) \mid$
			$\Pi_{\textsc{Upd}}\{\!\{B, k\}\!\}(a, \varsigma(x)b)$
naked methods	B	::=	$\overrightarrow{\varsigma(\overrightarrow{y})b}$
advice	$\mathcal{A}$	::=	$\mathit{pcd} \triangleright \varsigma(\overrightarrow{y})b$
step kinds	ρ	::=	$\textsc{Val} \mid \textsc{Ivk} \mid \textsc{Upd}$

Meta-syntax:

reduction judgments			$\mathcal{K} \vdash_{\boldsymbol{M}, \overrightarrow{\mathcal{A}}} a \rightsquigarrow v$
evaluation contexts	$\mathcal{K}$	::=	$\epsilon \mid \kappa \cdot \mathcal{K}$
evaluation steps	κ	::=	$\mathsf{ib}(\overline{l}, l) \mid \mathsf{va} \mid \mathsf{ia} \mid \mathsf{ua}$

Figure 1: Syntax and meta-syntax of $\varsigma_{asp}(M)$, where $\mathcal{C}$ is the syntax of M's point cut description language

Programs, denoted by $\mathcal{P}$, are pairs, $a \otimes \overrightarrow{\mathcal{A}}$, consisting of a single term and a sequence of advice. The term represents the base program to be evaluated.

In the syntax, a, b, and c range over terms. To the term syntax of Abadi and Cardelli's ς calculus, [1, pp. 57], we add $\mathsf{proceed}$ terms for continuing from advice code to the code which it advises (or any lower precedence advice). We also add basic and functional constants.

The basic terms are variable references (like x), values, and value proceed terms, written $\mathsf{proceed}_{\textsc{Val}}()$, for continuing from advice on values. A value may be a basic constant, represented by d, or an object, written $[\overline{l_i = \varsigma(x_i)b_i}^{\,i \in I}]$. An object is a set of labeled methods. Methods have the form $\varsigma(x)b$, where x represents the "self" parameter and b is a term. Since an object is a set of labeled methods, we equate objects when they only differ by a reordering of methods. We also equate methods when they only differ up to alpha congruence (i.e., renaming of self parameters).

The composite terms are: $a.k$, the selection of a method or application of a functional constant, together referred to as *invocation*; $a.l \Leftarrow \varsigma(x)b$, method update, for changing the method to which a label is bound; $\mathsf{proceed}_{\textsc{Ivk}}(a)$, invocation proceed, for continuing from advice on invocation; and $\mathsf{proceed}_{\textsc{Upd}}(a, \varsigma(x)b)$, update proceed, for continuing from advice on method update.

The calculus includes another set of terms called *proceed closures*. Proceed closures are created dynamically by the semantics during the evaluation of advice. Proceed terms appearing in the body of advice are replaced with proceed

closures. These closures carry a *thunk* that tracks the advised code and any lower-precedence advice. This tracking allows the semantics to continue to the lower-precedence advice, and ultimately to the advised code. Although proceed closures are part of the term syntax, they are not allowed in user programs; they may only be dynamically generated by the semantics. A simple check on user programs can ensure this.

An advice sequence $\overrightarrow{\mathcal{A}}$ in $\varsigma_{asp}(\boldsymbol{M})$ associates point cut descriptions with advice. A single piece of advice in $\overrightarrow{\mathcal{A}}$ has the form $pcd \triangleright \varsigma(\overrightarrow{y})b$, where pcd denotes an arbitrary point cut description drawn from $\boldsymbol{M}$'s syntax, $\boldsymbol{\mathcal{C}}$. The advice body b is applied at the points in a reduction that match pcd according to the point cut description language's *match* function. The bound variables in the advice body are given by $\overrightarrow{y}$. The number of bound variables depends on whether the advice applies to values, invocation, or method update. The use of these bound variables is described below.

Figure 1 also gives the meta-syntax used in the semantics. A judgment of the form $\mathcal{K} \vdash_{\boldsymbol{M},\overrightarrow{\mathcal{A}}} a \rightsquigarrow v$ says that the term a reduces to the value v in the evaluation context $\mathcal{K}$, given the point cut description language $\boldsymbol{M}$ and the advice sequence $\overrightarrow{\mathcal{A}}$. We omit $\boldsymbol{M}$ and $\overrightarrow{\mathcal{A}}$ from the notation when they are clear from context. The *evaluation context*, $\mathcal{K}$, is an encoding of all the reduction steps below the judgment in a proof tree. An evaluation context corresponds to the call stack in a dynamic aspect-oriented language like AspectJ.

2.1.2 Helper Functions

The reduction rules described in subsequent subsections use three helper functions. The advice lookup function for a point cut description language $\boldsymbol{M} = \langle \boldsymbol{\mathcal{C}}, match \rangle$ is defined recursively:

$$advFor_{\boldsymbol{M}}(jp, \bullet) = \bullet$$

$$advFor_{\boldsymbol{M}}(jp, (pcd \triangleright \varsigma(\overrightarrow{y})b) + \overrightarrow{\mathcal{A}}) = match(pcd \triangleright \varsigma(\overrightarrow{y})b, jp) + advFor_{\boldsymbol{M}}(jp, \overrightarrow{\mathcal{A}})$$

The signature function returns a value's signature:

$$sig(v) = \begin{cases} \overline{l_i}^{\,i \in I} & \text{if } v = [\overline{l_i = \varsigma(x_i)b_i}^{\,i \in I}] \\ v & \text{otherwise} \end{cases}$$

The $close_\rho$ function takes a term, representing an advice body, and some additional information, and produces another term in which all the $\mathsf{proceed}_\rho$ sub-terms have been converted into proceed closures. For $\mathsf{proceed}$ terms the function is defined as follows:

$$close_{\text{VAL}}(\mathsf{proceed}_{\text{VAL}}(), \{\!|B, v|\!\}) = \Pi_{\text{VAL}}\{\!|B, v|\!\}()$$

$$close_{\text{IVK}}(\mathsf{proceed}_{\text{IVK}}(a), \{\!|B, S, k|\!\}) = \Pi_{\text{IVK}}\{\!|B, S, k|\!\}(close_{\text{IVK}}(a, \{\!|B, S, k|\!\}))$$

$$close_{\text{UPD}}(\mathsf{proceed}_{\text{UPD}}(a, \varsigma(x)b), \{\!|B, k|\!\}) = \Pi_{\text{UPD}}\{\!|B, k|\!\}(close_{\text{UPD}}(a, \{\!|B, k|\!\}), \varsigma(x) close_{\text{UPD}}(b, \{\!|B, k|\!\}))$$

For proceed closures, the function is undefined; proceed closures may not appear in advice bodies. For all other terms, the function simply recurses on sub-terms [6].

2.1.3 Rules When No Advice Matches

The rules RED VAL 0, RED SEL 0, and RED UPD 0, given in Figure 2, correspond to the three reduction rules in the operational semantics for the functional ς calculus [1, p. 64], with RED VAL 0 generalized to handle both objects and basic constants. In each of these rules there is a premise that asserts that advice lookup yields the empty sequence. Since these rules correspond to those for the ς calculus, we omit examples here, however, three details bear mentioning:

$$\text{RED VAL 0} \quad \frac{\mathcal{K} \vdash_{\boldsymbol{M},\overrightarrow{\mathcal{A}}} \diamond \qquad advFor_{\boldsymbol{M}}(\langle \text{VAL}, \mathcal{K}, sig(v), \epsilon \rangle, \overrightarrow{\mathcal{A}}) = \bullet}{\mathcal{K} \vdash_{\boldsymbol{M},\overrightarrow{\mathcal{A}}} v \rightsquigarrow v}$$

$$\text{RED SEL 0 (where } o \triangleq [\overline{l_i = \varsigma(x_i)b_i}^{\,i \in I}]) \quad \frac{\mathcal{K} \vdash_{\boldsymbol{M},\overrightarrow{\mathcal{A}}} a \rightsquigarrow o \qquad l_j \in \overline{l_i}^{\,i \in I} \qquad advFor_{\boldsymbol{M}}(\langle \text{IVK}, \mathcal{K}, \overline{l_i}^{\,i \in I}, l_j \rangle, \overrightarrow{\mathcal{A}}) = \bullet \qquad \mathsf{ib}(\overline{l_i}^{\,i \in I}, l_j) \cdot \mathcal{K} \vdash_{\boldsymbol{M},\overrightarrow{\mathcal{A}}} b_j\{\!|x_j \leftarrow o|\!\} \rightsquigarrow v}{\mathcal{K} \vdash_{\boldsymbol{M},\overrightarrow{\mathcal{A}}} a.l_j \rightsquigarrow v}$$

$$\text{RED UPD 0 (where } o \triangleq [\overline{l_i = \varsigma(x_i)b_i}^{\,i \in I}]) \quad \frac{\mathcal{K} \vdash_{\boldsymbol{M},\overrightarrow{\mathcal{A}}} a \rightsquigarrow o \qquad l_j \in \overline{l_i}^{\,i \in I} \qquad advFor_{\boldsymbol{M}}(\langle \text{UPD}, \mathcal{K}, \overline{l_i}^{\,i \in I}, l_j \rangle, \overrightarrow{\mathcal{A}}) = \bullet}{\mathcal{K} \vdash_{\boldsymbol{M},\overrightarrow{\mathcal{A}}} a.l_j \Leftarrow \varsigma(x)b \rightsquigarrow [\overline{l_i = \varsigma(x_i)b_i}^{\,i \in I \setminus \{j\}}, l_j = \varsigma(x)b]}$$

$$\text{RED FCONST 0} \quad \frac{\mathcal{K} \vdash_{\boldsymbol{M},\overrightarrow{\mathcal{A}}} a \rightsquigarrow v' \qquad advFor_{\boldsymbol{M}}(\langle \text{IVK}, \mathcal{K}, sig(v'), f \rangle, \overrightarrow{\mathcal{A}}) = \bullet \qquad \mathsf{ib}(sig(v'), f) \cdot \mathcal{K} \vdash_{\boldsymbol{M},\overrightarrow{\mathcal{A}}} \delta(f, v') \rightsquigarrow v}{\mathcal{K} \vdash_{\boldsymbol{M},\overrightarrow{\mathcal{A}}} a.f \rightsquigarrow v}$$

Figure 2: Reduction rules when no advice matches

- In RED VAL 0, the premise $\mathcal{K} \vdash_{\boldsymbol{M},\overrightarrow{\mathcal{A}}} \diamond$ indicates that the environment is well formed.
- In RED SEL 0, when reducing the method body, b, the semantics prefixes $\mathsf{ib}(\overline{l_i}^{\,i \in I}, l_j)$ to the evaluation context. By recording every invocation during a reduction, the evaluation context models the call stack in a typical language implementation. This is useful to model constructs like AspectJ's cflow point cuts.
- We write $a\{\!|x \leftarrow b|\!\}$ to denote the standard, capture-avoiding, substitution of b for x in a.

A companion technical report provides the formal definition of these and other details suppressed below, as well as detailed reductions for all examples [6].

The rule RED FCONST 0, also given in Figure 2, specifies the semantics of functional constants when no advice matches. The rule says that when a functional constant is applied to a term, the term is first reduced to a value and then the δ function is applied to the functional constant and value to yield the final result. Thus, the δ function, which is intentionally underspecified here, determines the meaning of the functional constants. We restrict δ to just depend on observable characteristics of v [6].

Through the remainder of this paper we will use examples based on the point object $[\mathsf{n}{=}\varsigma(\mathsf{y})\ 0,\ \mathsf{pos}{=}\varsigma(\mathsf{p})\ \mathsf{p.n}]$. In the example, n is a "field" storing the point's position and pos is a method for querying the point's position.

2.1.4 Rules When Some Advice Matches

$$\textsc{Red Val 1}\quad \frac{\mathcal{K} \vdash_{\mathcal{M},\overrightarrow{\mathcal{A}}} \diamond \qquad \mathit{advFor}_{\mathcal{M}}(\langle \textsc{Val}, \mathcal{K}, \mathit{sig}(v), \epsilon\rangle, \overrightarrow{\mathcal{A}}) = \varsigma()b + B \qquad \mathit{close}_{\textsc{Val}}(b, \{\!|B, v|\!\}) = b' \qquad \mathsf{va} \cdot \mathcal{K} \vdash_{\mathcal{M},\overrightarrow{\mathcal{A}}} b' \rightsquigarrow v'}{\mathcal{K} \vdash_{\mathcal{M},\overrightarrow{\mathcal{A}}} v \rightsquigarrow v'}$$

$$\textsc{Red Sel 1}\ (\text{where } o \triangleq [\overline{l_i = \varsigma(x_i)b_i}^{\,i\in I}])\quad \frac{\mathcal{K} \vdash_{\mathcal{M},\overrightarrow{\mathcal{A}}} a \rightsquigarrow o \qquad l_j \in \overline{l_i}^{\,i\in I} \qquad \mathit{advFor}_{\mathcal{M}}(\langle \textsc{Ivk}, \mathcal{K}, \overline{l_i}^{\,i\in I}, l_j\rangle, \overrightarrow{\mathcal{A}}) = \varsigma(y)b + B \qquad \mathit{close}_{\textsc{Ivk}}(b, \{\!|(B + \varsigma(x_j)b_j), \overline{l_i}^{\,i\in I}, l_j|\!\}) = b' \qquad \mathsf{ia} \cdot \mathcal{K} \vdash_{\mathcal{M},\overrightarrow{\mathcal{A}}} b'\{\!\{y \leftarrow o\}\!\} \rightsquigarrow v}{\mathcal{K} \vdash_{\mathcal{M},\overrightarrow{\mathcal{A}}} a.l_j \rightsquigarrow v}$$

$$\textsc{Red FConst 1}\quad \frac{\mathcal{K} \vdash_{\mathcal{M},\overrightarrow{\mathcal{A}}} a \rightsquigarrow v' \qquad \mathit{advFor}_{\mathcal{M}}(\langle \textsc{Ivk}, \mathcal{K}, \mathit{sig}(v'), f\rangle, \overrightarrow{\mathcal{A}}) = \varsigma(y)b + B \qquad \mathit{close}_{\textsc{Ivk}}(b, \{\!|B, \mathit{sig}(v'), f|\!\}) = b' \qquad \mathsf{ia} \cdot \mathcal{K} \vdash_{\mathcal{M},\overrightarrow{\mathcal{A}}} b'\{\!\{y \leftarrow v'\}\!\} \rightsquigarrow v}{\mathcal{K} \vdash_{\mathcal{M},\overrightarrow{\mathcal{A}}} a.f \rightsquigarrow v}$$

$$\textsc{Red Upd 1}\ (\text{where } o \triangleq [\overline{l_i = \varsigma(x_i)b_i}^{\,i\in I}])\quad \frac{\mathcal{K} \vdash_{\mathcal{M},\overrightarrow{\mathcal{A}}} a \rightsquigarrow o \qquad \mathit{advFor}_{\mathcal{M}}(\langle \textsc{Upd}, \mathcal{K}, \overline{l_i}^{\,i\in I}, l_j\rangle, \overrightarrow{\mathcal{A}}) = \varsigma(\mathit{targ}, \mathit{rval})b' + B \qquad \mathit{close}_{\textsc{Upd}}(b', \{\!|B, l_j|\!\}) = b'' \qquad \mathsf{ua} \cdot \mathcal{K} \vdash_{\mathcal{M},\overrightarrow{\mathcal{A}}} b''\{\!\{\mathit{rval} \leftarrow b\{\!\{x \leftarrow \mathit{targ}\}\!\}\}\!\}_{\mathit{targ}}\{\!\{\mathit{targ} \leftarrow o\}\!\} \rightsquigarrow v}{\mathcal{K} \vdash_{\mathcal{M},\overrightarrow{\mathcal{A}}} a.l_j \Leftarrow \varsigma(x)b \rightsquigarrow v}$$

Figure 3: Reduction rules when some advice matches

The rules in Figure 3 correspond to the basic operations when there is applicable advice. In addition to invocation, these rules provide join points for values and method update. These additional join points are unique among formal models of aspect-oriented languages, and are used to model the AspectJ's open classes, HyperJ, and adaptive methods, as discussed in Section 3.

For objects and basic constants, Red Val 1 constructs a join point, $\langle \textsc{Val}, \mathcal{K}, \mathit{sig}(v), \epsilon\rangle$, that contains the signature of the value. This join point is passed to the advice lookup function, which must return a non-empty sequence of advice for this rule to be applicable.[1] The rule closes any $\mathsf{proceed}_{\textsc{Val}}$ sub-terms in the body, b, of the first element in the advice sequence, yielding a term b'. The $\mathit{close}_{\textsc{Val}}$ function uses the thunk, $\{\!|B, v|\!\}$, which stores any other applicable advice, B, and the original value, v. See Section 2.1.5 for how this information is used for proceeding from value advice. Finally, the rule reduces b' in an environment whose evaluation context is extended with va, indicating that the reduction is value advice. The result, v', of reducing b' is the result of the advised value term.

For method selection, Red Sel 1 reduces the target term, a, to an object value, o, and verifies, by the $l_j \in \overline{l_i}^{\,i\in I}$ premise, that the selected method is defined in the target object. The rule constructs a join point, $\langle \textsc{Ivk}, \mathcal{K}, \overline{l_i}^{\,i\in I}, l_j\rangle$, that contains the signature of the target object and the label of the method to be selected. This join point is passed to the advice lookup function, which must return a non-empty sequence of advice for this rule to be applicable.

Advice for method selection, $\varsigma(y)b$, has one parameter. The rule closes any $\mathsf{proceed}_{\textsc{Ivk}}$ sub-terms in the body, b, of the first element in the advice sequence, yielding a term b'. The thunk, $\{\!|(B + \varsigma(x_j)b_j), \overline{l_i}^{\,i\in I}, l_j|\!\}$, records any other applicable advice plus the originally selected method (as $B + \varsigma(x_j)b_j$), the set of labels in the target object, and the selected label. The rule then reduces b', substituting the target object, o, for each instance of the parameter, y. The rule extends the evaluation context, adding ia to indicate that the reduction of b' is invocation advice. The result, v, of reducing the advice is the result of the entire rule.

Red FConst 1 only differs slightly from Red Sel 1. The functional constant rule allows the target term to reduce to any value, not just an object.

Red Upd 1, for method update, is also similar to Red Sel 1. The key differences are that advice on method update, $\varsigma(\mathit{targ}, \mathit{rval})b'$, has two parameters and the reduction of the closed advice body, b'', uses both capture-avoiding and capturing substitution. The first advice parameter, *targ*, corresponds to the target object, o, of the update operation. The second parameter, *rval*, corresponds to the body, b, of the update's r-value. The combination of capturing and capture-avoiding substitution in the reduction rule allows some interesting tricks. We illustrate this with examples below, after first giving more details on the two sorts of substitution.

In general, substitution of c for x in b, with capture of any free occurrences of z in c, is written $b\{\!\{x \hookleftarrow c\}\!\}_z$. The only difference between capture-avoiding and capturing substitution is in the definition for substitution under variable binders. For capture-avoiding substitution, the definition is:

$$(\varsigma(y)b)\{\!\{x \leftarrow c\}\!\} \triangleq \varsigma(y')(b\{\!\{y \leftarrow y'\}\!\}\{\!\{x \leftarrow c\}\!\})$$

where $y' \notin FV(\varsigma(y)b) \cup FV(c) \cup \{x\}$. The renaming of y to a fresh name y' prevents capture. On the other hand, for capturing substitution, there are two cases:

$$\begin{array}{lcl} (\varsigma(z)b)\{\!\{x \hookleftarrow c\}\!\}_z & \triangleq & \varsigma(z)(b\{\!\{x \hookleftarrow c\}\!\}_z) \\ (\varsigma(y)b)\{\!\{x \hookleftarrow c\}\!\}_z & \triangleq & \varsigma(y')(b\{\!\{y \leftarrow y'\}\!\}\{\!\{x \hookleftarrow c\}\!\}_z) \text{ if } y \neq z \end{array}$$

The first case captures free instances of z in c by omitting the renaming step. The second case avoids capture when the bound variable, y, in the term differs from the variable, z, to be captured by the substitution.

The combination of the two sorts of substitution in Red Upd 1 gives advice authors some interesting options. The first substitution, $b\{\!\{x \leftarrow \mathit{targ}\}\!\}$, renames the self parameter in the body, b, of the original r-value. The next substitution is the *targ*-capturing substitution for *rval* in the advice body, b''. This capturing substitution allows the advice author to capture occurrences of the self-parameter, by placing *rval* under a $\varsigma(\mathit{targ})$ binder, or not capture occurrences of the self-parameter, by not placing *rval* under a binder or by placing it under a non-*targ* binder. The following two examples illustrate the utility of combining the two sorts of substitution. Consider the update:

$$\mathsf{[n{=}\varsigma(y)0,\ pos{=}\varsigma(p)p.n].pos} \Leftarrow \mathsf{\varsigma(x)x.n.succ}$$

In the absence of advice, this would reduce to:

$$\mathsf{[n{=}\varsigma(y)0,\ pos{=}\varsigma(x)x.n.succ]}$$

[1] In Red Val 1, and the other "1" rules, if any of the elements of the sequence returned by advice lookup has the wrong number of parameters, then the reduction may stick.

We first consider advice that fixes the value of the pos method to the result of evaluating the new method body, x.n.succ, substituting the original target object for x:

$$\varsigma(\mathsf{targ},\mathsf{rval})\mathsf{proceed}_{\text{UPD}}(\mathsf{targ},\ \varsigma(\mathsf{z})\mathsf{rval})$$

Assuming no other advice was found in the advice lookup, Red Upd 1 closes the $\mathsf{proceed}_{\text{UPD}}$ sub-term in the advice body using the thunk $\{\!|\bullet,\mathsf{pos}|\!\}$. Thus, we get:

$$b'' = \Pi_{\text{UPD}}\{\!|\bullet,\mathsf{pos}|\!\}(\mathsf{targ},\ \varsigma(\mathsf{z})\mathsf{rval})$$

We now consider the substitutions in the last premise of Red Upd 1. Because targ does not appear bound in b'', there is no capture in these substitutions. After making the replacements specified in the rule, the substitutions are carried out as follows (where underlining shows the subterm that is the target of the next substitution):

$$\begin{aligned}
&\Pi_{\text{UPD}}\{\!|\bullet,\mathsf{pos}|\!\}(\mathsf{targ},\ \varsigma(\mathsf{z})\mathsf{rval})\{\!|\mathsf{rval} \hookleftarrow \underline{\mathsf{x.n.succ}}\{\!|\mathsf{x} \leftarrow \mathsf{targ}|\!\}|\!\}_{\mathsf{targ}} \\
&\qquad\qquad \{\!|\mathsf{targ} \leftarrow [\mathsf{n}{=}\varsigma(\mathsf{y})0,\ \mathsf{pos}{=}\varsigma(\mathsf{p})\mathsf{p.n}]|\!\} \\
&= \underline{\Pi_{\text{UPD}}\{\!|\bullet,\mathsf{pos}|\!\}(\mathsf{targ},\ \varsigma(\mathsf{z})\mathsf{rval})}\{\!|\mathsf{rval} \hookleftarrow \mathsf{targ.n.succ}|\!\}_{\mathsf{targ}} \\
&\qquad\qquad \{\!|\mathsf{targ} \leftarrow [\mathsf{n}{=}\varsigma(\mathsf{y})0,\ \mathsf{pos}{=}\varsigma(\mathsf{p})\mathsf{p.n}]|\!\} \\
&= \underline{\Pi_{\text{UPD}}\{\!|\bullet,\mathsf{pos}|\!\}(\mathsf{targ},\ \varsigma(\mathsf{z})\mathsf{targ.n.succ})} \\
&\qquad\qquad \{\!|\mathsf{targ} \leftarrow [\mathsf{n}{=}\varsigma(\mathsf{y})0,\ \mathsf{pos}{=}\varsigma(\mathsf{p})\mathsf{p.n}]|\!\} \\
&= \Pi_{\text{UPD}}\{\!|\bullet,\mathsf{pos}|\!\}([\mathsf{n}{=}\varsigma(\mathsf{y})0,\ \mathsf{pos}{=}\varsigma(\mathsf{p})\mathsf{p.n}], \\
&\qquad\qquad \varsigma(\mathsf{z})[\mathsf{n}{=}\varsigma(\mathsf{y})0,\ \mathsf{pos}{=}\varsigma(\mathsf{p})\mathsf{p.n}].\mathsf{n.succ})
\end{aligned}$$

The last term will reduce (as discussed in Section 2.1.5) to:

$$[\mathsf{n}{=}\varsigma(\mathsf{y})0,\ \mathsf{pos}{=}\varsigma(\mathsf{z})[\mathsf{n}{=}\varsigma(\mathsf{y})0,\ \mathsf{pos}{=}\varsigma(\mathsf{p})\mathsf{p.n}].\mathsf{n.succ}]$$

where the original object is now embedded in the pos method. No matter what we set n to in this term, selecting pos on this term will yield the successor of 0. Because no capture occurred, the advice has fixed the value of pos to that obtained by calling the updated method at the time of the update.[2]

As a second example, consider advice that uses the body of the update's r-value without causing it to be reduced. Suppose the advice lookup in the Red Upd 1 rule returned:

$$\varsigma(\mathsf{targ},\mathsf{rval})\mathsf{proceed}_{\text{UPD}}(\mathsf{targ},\varsigma(\mathsf{targ})\mathsf{rval.succ})$$

Assuming no other advice was found in the advice lookup, then after closing the $\mathsf{proceed}_{\text{UPD}}$ sub-term, the substitutions for this advice are:

$$\begin{aligned}
&\Pi_{\text{UPD}}\{\!|\bullet,\mathsf{pos}|\!\}(\mathsf{targ},\varsigma(\mathsf{targ})\mathsf{rval.succ}) \\
&\qquad \{\!|\mathsf{rval} \hookleftarrow \underline{\mathsf{x.n.succ}}\{\!|\mathsf{x} \leftarrow \mathsf{targ}|\!\}|\!\}_{\mathsf{targ}} \\
&\qquad\qquad \{\!|\mathsf{targ} \leftarrow [\mathsf{n}{=}\varsigma(\mathsf{y})0,\ \mathsf{pos}{=}\varsigma(\mathsf{p})\mathsf{p.n}]|\!\} \\
&= \underline{\Pi_{\text{UPD}}\{\!|\bullet,\mathsf{pos}|\!\}(\mathsf{targ},\varsigma(\mathsf{targ})\mathsf{rval.succ})} \\
&\qquad\qquad \{\!|\mathsf{rval} \hookleftarrow \mathsf{targ.n.succ}|\!\}_{\mathsf{targ}} \\
&\qquad\qquad \{\!|\mathsf{targ} \leftarrow [\mathsf{n}{=}\varsigma(\mathsf{y})0,\ \mathsf{pos}{=}\varsigma(\mathsf{p})\mathsf{p.n}]|\!\} \\
&=^{\dagger} \underline{\Pi_{\text{UPD}}\{\!|\bullet,\mathsf{pos}|\!\}(\mathsf{targ},\varsigma(\mathsf{targ})\mathsf{targ.n.succ.succ})} \\
&\qquad\qquad \{\!|\mathsf{targ} \leftarrow [\mathsf{n}{=}\varsigma(\mathsf{y})0,\ \mathsf{pos}{=}\varsigma(\mathsf{p})\mathsf{p.n}]|\!\} \\
&= \Pi_{\text{UPD}}\{\!|\bullet,\mathsf{pos}|\!\}([\mathsf{n}{=}\varsigma(\mathsf{y})0,\ \mathsf{pos}{=}\varsigma(\mathsf{p})\mathsf{p.n}], \\
&\qquad\qquad \varsigma(\mathsf{targ})\mathsf{targ.n.succ.succ})
\end{aligned}$$

[2]Although our calculus is functional, the order of execution matters. The semantics is essentially call-by-value; the target sub-term of an operation is evaluated to a value before the operation takes place.

This term will reduce to:

$$[\mathsf{n}{=}\varsigma(\mathsf{y})0,\ \mathsf{pos}{=}\varsigma(\mathsf{targ})\mathsf{targ.n.succ.succ}]$$

Thus, the advice causes any subsequent selections of pos to return the successor of what they would have if the update had happened without advice. The substitution that captures the free targ in targ.n.succ, flagged with a dagger (†), is the key to this behavior. The capture allows the body of the r-value in the update to be syntactically embedded inside another term whose evaluation is suspended by a binder.

2.1.5 Proceeding from Advice

Figure 4 gives the reduction rules for proceeding from advice. The rules only reduce proceed closures. Proceed terms that are not closed, such as those appearing outside of advice, are errors (and cause reductions to stick).

There are four pairs of rules, one pair each for proceeding from advice on values, method selection, functional constant application, and method update. The "1" rule in each pair proceeds to the advice with the next highest precedence; the "0" rule proceeds to the original operation when no lower-precedence advice remains.

When proceeding from invocation or update advice, the target object is found by reducing the first term in the proceed closure. When proceeding from update advice, the r-value is also taken from the proceed closure. The proceed closure for value advice does not give a new term; if it did, and if the originally advised term appeared in the new term, then this would re-trigger the advice, causing evaluation to loop.

The four "0" rules are similar to the corresponding rules in Figure 2 for operations with no matching advice. One interesting detail is that Red SPrcd 0 extracts the original method, $\varsigma(y)b$, from the proceed closure, where it was stored by Red Sel 1, instead of extracting the method from the new target object, o. This permits advice to execute the code defined in the original object while using different data.

The four "1" rules are similar to the corresponding rules in Figure 3 for operations with matching advice, but instead of calling the advice lookup function to find applicable advice, they extract the next advice from the proceed closure.

To illustrate the use of proceed, we introduce a simple point cut description language that allows advice on method selection. Let $\boldsymbol{M}_s = \langle \boldsymbol{\mathcal{C}}_s, match_s \rangle$, where $\boldsymbol{\mathcal{C}}_s ::= [\bar{l}].l$ and:

$$match_s([\bar{l}].l \rhd \varsigma(\overrightarrow{y})b, \langle \rho, \mathcal{K}, S, k\rangle) = \begin{cases} \langle \varsigma(\overrightarrow{y})b\rangle & \text{if } (\rho = \text{Ivk}) \wedge (S = \bar{l}) \wedge (k = l) \\ \bullet & \text{otherwise} \end{cases}$$

In $\varsigma_{asp}(\boldsymbol{M}_s)$, proceed can be used to encode before, after, and around advice as in AspectJ.

Before advice, or advice that is executed before the body of a method, can be written by changing the target object before proceeding. For example, let

$$\boldsymbol{A} \triangleq [\mathsf{n},\mathsf{pos}].\mathsf{pos} \rhd \varsigma(\mathsf{x})\mathsf{proceed}_{\text{IvK}}(\mathsf{x.n} \Leftarrow \varsigma(\mathsf{y})0)$$

This advice advises selection on the pos method so that before reducing the body of that method, the n field of the target object is set to 0. That is, Red SPrcd 0 evaluates $\mathsf{x.n} \Leftarrow \varsigma(\mathsf{y})0$ before evaluating the original body of the pos method. Thus, without advice

$$\epsilon \vdash_{\boldsymbol{M}_s,\bullet} [\mathsf{n}{=}\varsigma(\mathsf{y})2,\mathsf{pos}{=}\varsigma(\mathsf{p})\mathsf{p.n}].\mathsf{pos} \rightsquigarrow 2,$$

$$\textsc{Red VPrcd 0}\quad \frac{\mathcal{K} \vdash_{M,\overline{\mathcal{A}}} \diamond}{\mathcal{K} \vdash_{M,\overline{\mathcal{A}}} \Pi_{\textsc{Val}}\{\!|\bullet, v|\!\}() \rightsquigarrow v}$$

$$\textsc{Red VPrcd 1}\quad \frac{\mathcal{K} \vdash_{M,\overline{\mathcal{A}}} \diamond \qquad close_{\textsc{Val}}(b, \{\!|B, v|\!\}) = b' \qquad \mathsf{va} \cdot \mathcal{K} \vdash_{M,\overline{\mathcal{A}}} b' \rightsquigarrow v'}{\mathcal{K} \vdash_{M,\overline{\mathcal{A}}} \Pi_{\textsc{Val}}\{\!|(\varsigma()b + B), v|\!\}() \rightsquigarrow v'}$$

$$\textsc{Red SPrcd 0}\quad \frac{\mathcal{K} \vdash_{M,\overline{\mathcal{A}}} a \rightsquigarrow o \qquad \mathsf{ib}(\bar{l}, l) \cdot \mathcal{K} \vdash_{M,\overline{\mathcal{A}}} b\{\!|y \leftarrow o|\!\} \rightsquigarrow v}{\mathcal{K} \vdash_{M,\overline{\mathcal{A}}} \Pi_{\textsc{Ivk}}\{\!|\varsigma(y)b, \bar{l}, l|\!\}(a) \rightsquigarrow v}$$

$$\textsc{Red SPrcd 1}\quad \frac{\mathcal{K} \vdash_{M,\overline{\mathcal{A}}} a \rightsquigarrow o \qquad B \neq \bullet \qquad close_{\textsc{Ivk}}(b, \{\!|B, \bar{l}, l|\!\}) = b' \qquad \mathsf{ia} \cdot \mathcal{K} \vdash_{M,\overline{\mathcal{A}}} b'\{\!|y \leftarrow o|\!\} \rightsquigarrow v}{\mathcal{K} \vdash_{M,\overline{\mathcal{A}}} \Pi_{\textsc{Ivk}}\{\!|(\varsigma(y)b + B), \bar{l}, l|\!\}(a) \rightsquigarrow v}$$

$$\textsc{Red FPrcd 0}\quad \frac{\mathcal{K} \vdash_{M,\overline{\mathcal{A}}} a \rightsquigarrow v' \qquad \mathsf{ib}(S, f) \cdot \mathcal{K} \vdash_{M,\overline{\mathcal{A}}} \delta(f, v') \rightsquigarrow v}{\mathcal{K} \vdash_{M,\overline{\mathcal{A}}} \Pi_{\textsc{Ivk}}\{\!|\bullet, S, f|\!\}(a) \rightsquigarrow v}$$

$$\textsc{Red FPrcd 1}\quad \frac{\mathcal{K} \vdash_{M,\overline{\mathcal{A}}} a \rightsquigarrow v' \qquad close_{\textsc{Ivk}}(b, \{\!|B, S, f|\!\}) = b' \qquad \mathsf{ia} \cdot \mathcal{K} \vdash_{M,\overline{\mathcal{A}}} b'\{\!|y \leftarrow v'|\!\} \rightsquigarrow v}{\mathcal{K} \vdash_{M,\overline{\mathcal{A}}} \Pi_{\textsc{Ivk}}\{\!|(\varsigma(y)b + B), S, f|\!\}(a) \rightsquigarrow v}$$

$$\textsc{Red UPrcd 0}\quad \frac{\mathcal{K} \vdash_{M,\overline{\mathcal{A}}} a \rightsquigarrow [\overline{l_i = \varsigma(x_i)b_i}^{\,i \in I}] \qquad l_j \in \overline{l_i}^{\,i \in I}}{\mathcal{K} \vdash_{M,\overline{\mathcal{A}}} \Pi_{\textsc{Upd}}\{\!|\bullet, l_j|\!\}(a, \varsigma(x)b) \rightsquigarrow [\overline{l_i = \varsigma(x_i)b_i}^{\,i \in I \setminus j}, l_j = \varsigma(x)b]}$$

$$\textsc{Red UPrcd 1}\quad \frac{\mathcal{K} \vdash_{M,\overline{\mathcal{A}}} a \rightsquigarrow o \qquad close_{\textsc{Upd}}(b', \{\!|B, l_j|\!\}) = b'' \qquad \mathsf{ua} \cdot \mathcal{K} \vdash_{M,\overline{\mathcal{A}}} b''\{\!|rval \leftarrow b\{\!|x \leftarrow targ|\!\}|\!\}_{targ}\{\!|targ \leftarrow o|\!\} \rightsquigarrow v}{\mathcal{K} \vdash_{M,\overline{\mathcal{A}}} \Pi_{\textsc{Upd}}\{\!|(\varsigma(targ, rval)b' + B), l_j|\!\}(a, \varsigma(x)b) \rightsquigarrow v}$$

Figure 4: Reduction rules for proceeding from advice in $\varsigma_{asp}(\boldsymbol{M})$

but with the advice $\boldsymbol{A}$,

$$\epsilon \vdash_{M_s,\boldsymbol{A}} [\mathsf{n}{=}\varsigma(\mathsf{y})2, \mathsf{pos}{=}\varsigma(\mathsf{p})\mathsf{p.n}].\mathsf{pos} \rightsquigarrow 0.$$

Similarly, *after advice*, or advice that is executed after the body of a method, can be written by proceeding using the original target object and then manipulating the result:

$$[\mathsf{n},\mathsf{pos}].\mathsf{pos} \rhd \varsigma(\mathsf{x})\mathsf{proceed}_{\textsc{Ivk}}(\mathsf{x}).\mathsf{succ}$$

Around advice is simply a combination of these notions of before and after advice; that is, around advice includes some code that is executed before the body of the advised method and some code that is executed after.

3. MODELING EXISTING LANGUAGES

Because of the richness of the join point model in the semantics of ς_{asp} we can model several existing aspect-oriented languages. This result supports our contention that our calculus is suitable for studying the reasoning properties of aspect-oriented language. In this subsection we describe how we can model AspectJ [3] and HyperJ [14], and we sketch a model for adaptive methods à la DemeterJ [11].

descriptions	pcd	::=	$\textsc{Val} \mid \textsc{Ivk} \mid \textsc{Upd} \mid$
			$\mathsf{k} = k \mid \mathsf{S} = S \mid \mathsf{K} \in r \mid$
			$\neg pcd \mid pcd \wedge pcd \mid pcd \vee pcd$
context expr.	r	::=	$\epsilon \mid \mathsf{ib}(M, m) \mid \mathsf{va} \mid \mathsf{ia} \mid \mathsf{ua} \mid$
			$\mathbf{.} \mid r + r \mid rr \mid r\mathsf{*}$
signatures	M	::=	$d \mid \bar{l} \mid \mathbf{.}$
messages	m	::=	$f \mid l \mid \mathbf{.}$

Figure 5: Point Cut Syntax, $\boldsymbol{\mathcal{C}}_G$, for $\boldsymbol{M}_G$

3.1 AspectJ

In this section, we use an instance of ς_{asp} to model all of the aspect-oriented features of AspectJ [3], including open classes (in Section 3.1.2) and all relevant point cut descriptions (in Section 3.1.1).

We define a general point cut description language, $\boldsymbol{M}_G = \langle \boldsymbol{\mathcal{C}}_G, match_G \rangle$, in which the point cut descriptions are boolean combinations of queries over the elements of tuples from $\boldsymbol{J}$. The syntax, $\boldsymbol{\mathcal{C}}_G$, is given in Figure 5. For brevity, the formal definition of $match_G$ is omitted here in favor of an intuitive explanation [6].

- A Val, Ivk, or Upd point cut matches any join point of the corresponding kind.
- A $\mathsf{k} = k$ point cut matches any join point whose label or functional constant is k.
- A $\mathsf{S} = S$ point cut matches any join point whose target value has the signature S.
- A $\mathsf{K} \in r$ point cut matches any join point whose evaluation context is described by the regular expression r. In the regular expressions the symbol '$\mathbf{.}$' is a wildcard, matching any evaluation step, signature, label, or functional constant, depending on the context.[3]
- A $\neg pcd$ point cut matches any join point that is not matched by the point cut pcd.
- A $pcd_1 \wedge pcd_2$ point cut matches any join point that is matched by both pcd_1 and pcd_2.
- A $pcd_1 \vee pcd_2$ point cut matches any join point that is matched by either pcd_1 or pcd_2 (or both).

A few details of AspectJ are abstracted away in our model, but this does not impact its utility for theoretical work. Because of the "classless" and functional nature of our calculus, our model omits the class-based and imperative features of AspectJ. Also our calculus does not include a module system, so any references to Java packages are omitted from our model.

Several point cut descriptions in AspectJ match based on an object's type, given as a class or interface name, commensurate with Java's by-name typing discipline. Like Abadi and Cardelli, we use the set of labels appearing in an object to represent the object's type [1, pp. 79–92].

[3]Our use of regular expressions for matching the evaluation context is motivated by Sereni and de Moor [15].

Table 1: Modeling primitive AspectJ point cuts

AspectJ Point Cut	Modeled In $\varsigma_{asp}(\boldsymbol{M}_G)$
call(void Point.pos())	$\textsc{Ivk} \wedge \mathsf{S} = \{\mathsf{n},\mathsf{pos}\} \wedge \mathsf{k} = \mathsf{pos}$
call(Point.new())	$\textsc{Val} \wedge \mathsf{S} = \{\mathsf{n},\mathsf{pos}\}$
execution(void Point.pos())	$\textsc{Val} \wedge \mathsf{K} \in \mathsf{ib}(\{\mathsf{n},\mathsf{pos}\}, \mathsf{pos})\mathbf{.*}$
get(int Point.n)	$\textsc{Ivk} \wedge \mathsf{S} = \{\mathsf{n},\mathsf{pos}\} \wedge \mathsf{k} = \mathsf{n}$
set(int Point.n)	$\textsc{Upd} \wedge \mathsf{S} = \{\mathsf{n},\mathsf{pos}\} \wedge \mathsf{k} = \mathsf{n}$
adviceexecution()	$\mathsf{K} \in \mathbf{.*}(\mathsf{va} + \mathsf{ia} + \mathsf{ua})\mathbf{.*}$
within(Point)	$\mathsf{K} \in \mathsf{ib}(\{\mathsf{n},\mathsf{pos}\}, \mathbf{.})\mathbf{.*}$
withincode(Point.pos)	$\mathsf{K} \in \mathsf{ib}(\{\mathsf{n},\mathsf{pos}\}, \mathsf{pos})\mathbf{.*}$
cflow(Point.pos)	$\mathsf{K} \in \mathbf{.*}\mathsf{ib}(\{\mathsf{n},\mathsf{pos}\}, \mathsf{pos})\mathbf{.*}$
cflowbelow(Point.pos)	$\mathsf{K} \in \mathbf{.*\,.}\,\mathsf{ib}(\{\mathsf{n},\mathsf{pos}\}, \mathsf{pos})\mathbf{.*}$
this(Point)	$\mathsf{K} \in \mathsf{ib}(\{\mathsf{n},\mathsf{pos}\}, \mathbf{.})\mathbf{.*}$
target(Point)	$\mathsf{S} = \{\mathsf{n},\mathsf{pos}\}$

3.1.1 Point Cut Descriptions in AspectJ

AspectJ allows advice to be applied using an extensive point cut language. We have already shown how to model before, after and around advice (in Section 2.1.5). In this section we survey AspectJ's primitive point cut descriptions. For each, we show how it can be modeled in $\varsigma_{asp}(\boldsymbol{M}_G)$.

Table 1 lists all of the AspectJ primitive point cut descriptions that may be sensibly modeled in $\varsigma_{asp}(\boldsymbol{M}_G)$. The first column gives an example of each point cut in AspectJ; the last column gives a $\varsigma_{asp}(\boldsymbol{M}_G)$ model of the example.

The call join point for methods in AspectJ refers to the point in client code immediately before control passes to the invoked method. This point is analogous with the step in Red Sel 0 and 1 immediately before a method body is reduced. When reducing the method invocation $[\mathsf{n}{=}\varsigma(\mathsf{y})\mathsf{0}, \mathsf{pos}{=}\varsigma(\mathsf{p})\mathsf{p.n}].\mathsf{pos}$ the join point used for advice lookup in Red Sel 0 and 1 is $\langle \textsc{Ivk}, \mathcal{K}, \{\mathsf{n},\mathsf{pos}\}, \mathsf{pos}\rangle$. In $\boldsymbol{M}_G$, the point cut description $\textsc{Ivk} \wedge \mathsf{S} = \{\mathsf{n},\mathsf{pos}\} \wedge \mathsf{k} = \mathsf{pos}$, shown in the table, matches this join point. Any advice on this point cut description would cause Red Sel 1 to be used instead of Red Sel 0, thus triggering the evaluation of the advice at the appropriate step to model the call join point for methods. Because $\varsigma_{asp}(\boldsymbol{M}_G)$ does not include explicit constructors, the call join point for constructors in AspectJ is modeled by a point cut descriptor on object creation.

The execution join point for methods in AspectJ refers to the point in client code when an invoked method begins executing. In $\varsigma_{asp}(\boldsymbol{M}_G)$, when a method body is evaluated the reduction rules recurse on the target sub-term until a value, either an object or basic constant, is reached. This value is then used by all the subsequent reduction steps in the evaluation of the method body. Within the evaluation of the method body, only this first term will match the point cut description, $\textsc{Val} \wedge \mathsf{K} \in \mathsf{ib}(\{\mathsf{n},\mathsf{pos}\}, \mathsf{pos})\mathbf{.*}$, shown in Table 1.

We use the same model for AspectJ's get point cut description as for call, because methods and fields are unified in the object calculus. We model set like get, but use the update, instead of invocation, reduction kind.

AspectJ's adviceexecution point cut matches any join point where advice is executing. Our model for this in $\varsigma_{asp}(\boldsymbol{M}_G)$ is straightforward: if advice is executing, then one of va, ia, or ua must appear in the evaluation context.

The two AspectJ point cuts within and withincode match any join point associated with the code statically appearing in a given class or method, respectively. We model this in $\varsigma_{asp}(\boldsymbol{M}_G)$ by checking for the evaluation of the appropriate method body in the evaluation context. For within(Point), if $\mathsf{ib}(\{\mathsf{n},\mathsf{pos}\}, \mathbf{.})$ is at the head of the evaluation context, then the body of some method (note the wildcard) defined in a point object is being reduced. Furthermore, no code defined outside of such a method has been called, otherwise the ib term would not be at the head of the evaluation context. The model for withincode is similar, but a specific method is named instead of using a wildcard.

The cflow point cut in AspectJ matches any join point when the given method has begun executing and not yet finished. For the given example, this corresponds to the situation in $\varsigma_{asp}(\boldsymbol{M}_G)$ where the ib term matching the method is somewhere in the evaluation context.

We model cflowbelow similarly, but we require, with the extra . before the ib, that the context record something after the given joint point. This translation considers even join points in the execution of advice to be "below" the given join point. Another possible interpretation for "below" would be to consider code in the advice to not be "below" the joint point; only code declared in a base term would be consider "below" the advised method. For our example, this alternative interpretation would be modeled:

$$\mathsf{K} \in \mathbf{.*}\mathsf{ib}(\mathbf{.},\mathbf{.})\mathsf{ib}(\{\mathsf{n},\mathsf{pos}\}, \mathsf{pos})\mathbf{.*}$$

AspectJ's this and target point cuts come in two forms. In the first form, the point cut's argument is a type name. We can model this form in $\varsigma_{asp}(\boldsymbol{M}_G)$ as shown in Table 1.[4] In the second form of these point cuts, the argument is a variable that is bound for use in the advice body. The semantics of $\varsigma_{asp}(\boldsymbol{M}_G)$ always substitutes the target object for a variable in the advice body, implicitly supporting the variable-binding form of the target point cut. On the other hand, the variable-binding form of the this point cut cannot be modeled in $\varsigma_{asp}(\boldsymbol{M}_G)$ without changing the semantics, because advice might bind to some join point in the body of a method after substitution of the "this" object for the self parameter of the method. In such a case, because the semantics do not track the "this" object, there is no way to bind it in the advice body. However, it would be straightforward to add support for a variable-binding form of the this point cut. For example, the semantics could record the "this" object within the $\mathsf{ib}(S, k)$ evaluation context entry.

We omit some AspectJ point cuts from Table 1 that cannot be sensibly modeled in $\varsigma_{asp}(\boldsymbol{M}_G)$. Some of these are omitted because they refer to base-language features that are not present in the object calculus:

- the execution and withincode point cuts when applied to constructors,
- the initialization, preinitialization, and staticinitialization point cuts for field initializers,
- the handler point cut for exception handling, and
- the args point cut for binding method arguments other than the target object

The only other AspectJ point cut that we omit is the if point cut. This point cut allows programmers to evaluate boolean-valued expressions during join point matching. The

[4] In a language with neither subtyping nor subclassing, $\mathsf{this}(T)$ is equivalent to $\mathsf{within}(T)$.

point cut description language $\boldsymbol{M}_G$ does not readily model this, but we could define one that does. The simplest way of doing so would be to have a point cut description that takes a function as a parameter. The point cut description would match if the function applied to the current join point evaluated to true.

Our model of AspectJ's point cut descriptions is direct. All of the models presented in Table 1 can be generated by context-free translations from the AspectJ versions.

3.1.2 Open Classes in AspectJ

Open classes allow a programmer to extend the set of methods that may be invoked on an object without changing the code of the original class of that object [5, 13]. AspectJ allows a programmer to declare an additional supertype for a given type and add new methods to a class. Because $\varsigma_{asp}(\boldsymbol{M}_G)$ does not include declared subtyping, there is no direct analogue for the former feature. For the latter feature, advice on object creation allows us to create a wrapper object that adds methods to existing objects.

Consider the following AspectJ declaration which adds a color field to all Point class instances:

```
int Point.color = 0;
```

A model of this in $\boldsymbol{M}_G$ uses two pieces of advice:

$$(\textsc{Val} \wedge \mathsf{S} = \{\mathsf{n},\mathsf{pos}\}) \rhd \varsigma()$$
$$[\mathsf{orig}{=}\varsigma(\mathsf{s})\mathsf{proceed}_{\textsc{Val}}(),\ \mathsf{n}{=}\varsigma(\mathsf{s})\mathsf{s.orig.n},\ \mathsf{pos}{=}\varsigma(\mathsf{s})\mathsf{s.orig.pos},\ \mathsf{color}{=}\varsigma(\mathsf{s})0]$$

$$(\textsc{Upd} \wedge \mathsf{S} = \{\mathsf{orig},\mathsf{n},\mathsf{pos},\mathsf{color}\} \wedge (\mathsf{k} = \mathsf{n} \vee \mathsf{k} = \mathsf{pos})) \rhd$$
$$\varsigma(\mathsf{t},\mathsf{r}) \quad [\mathsf{orig}{=}\varsigma(\mathsf{s})\mathsf{proceed}_{\textsc{Upd}}(\mathsf{t.orig},\ \varsigma(\mathsf{t})\mathsf{r}),\ \mathsf{n}{=}\varsigma(\mathsf{s})\mathsf{s.orig.n},\ \mathsf{pos}{=}\varsigma(\mathsf{s})\mathsf{s.orig.pos},\ \mathsf{color}{=}\varsigma(\mathsf{s})\mathsf{t.color}]$$

The first piece of advice is applicable to every creation of a point in the base program. The original point object is stored in the field orig of the wrapper object. Methods n and pos redirect selection of these methods on the wrapper object to the appropriate methods of the original object. Finally, the color field exists only in the wrapper object.

But this first piece of advice is not the whole story. The second piece is needed to deal with updates to the n field of a point object. For example, without advice the term:

$$([\mathsf{n}{=}\varsigma(\mathsf{y})0,\ \mathsf{pos}{=}\varsigma(\mathsf{p})\mathsf{p.n}].\mathsf{n} \Leftarrow \varsigma(\mathsf{y})2).\mathsf{pos}$$

reduces to 2. However, with just the first piece of advice, the term reduces to 0. This is because the assignment to n updates the wrapper object, replacing the redirection method:

$$[\mathsf{orig}{=}\varsigma(\mathsf{s})\Pi_{\textsc{Val}}\{\!|\bullet, [\mathsf{n}{=}\varsigma(\mathsf{y})0,\ \mathsf{pos}{=}\varsigma(\mathsf{p})\mathsf{p.n}]|\!\}(),\ \mathsf{n}{=}\varsigma(\mathsf{y})2,\quad \textit{// no longer redirects to } \mathsf{orig},\ \mathsf{pos}{=}\varsigma(\mathsf{s})\mathsf{s.orig.pos},\ \mathsf{color}{=}\varsigma(\mathsf{s})0]$$

The second piece of advice avoids this problem by redirecting updates to point, so that they update the original point object within the wrapper.

We directly model open classes using $\varsigma_{asp}(\boldsymbol{M}_G)$. The set of labels in the original object and the methods to be added are all that is needed to generate the necessary advice.

3.2 HyperJ

With $\varsigma_{asp}(\boldsymbol{M}_G)$ we can also model "multi-dimensional separation of concerns", as embodied in HyperJ [14]. Contrary to AspectJ, which divides a program asymmetrically into base language terms plus aspects, HyperJ uses a set of modules, called "hyperslices", each of equal standing, along with a module-interconnection language that specifies how these hyperslices should be statically composed to form a single module. In $\varsigma_{asp}(\boldsymbol{M}_G)$, we model hyperslices using advice, maintaining the symmetry of the hyperslices. We also model hyperslice composition with advice.

Suppose we have two hyperslices. One defines a point object whose pos method multiples the position before returning it. The other defines a multiplier object. We use advice on basic constants to model these two hyperslices:

$$(\textsc{Val} \wedge \mathsf{S} = \mathsf{PointSlice}) \rhd \varsigma() \quad [\mathsf{n}{=}\varsigma(\mathsf{y})\ 0,\ \mathsf{pos}{=}\varsigma(\mathsf{p})\ \mathsf{p.n} \times \mathsf{p.mult},\ \mathsf{mult}{=}\varsigma(\mathsf{y})\ \mathsf{y.outer.mult},\ \mathsf{outer}{=}\varsigma(\mathsf{y})\ \mathsf{y}]$$

$$(\textsc{Val} \wedge \mathsf{S} = \mathsf{MultSlice}) \rhd \varsigma() \quad [\mathsf{mult}{=}\varsigma(\mathsf{y})\ 1,\ \mathsf{outer}{=}\varsigma(\mathsf{y})\ \mathsf{y}]$$

To represent an abstract method, m, in a hyperslice, we use $m = \varsigma(\mathsf{y})\mathsf{y.outer}.m$, as in the mult method of PointSlice.[5]

Now suppose we want to compose these two hyperslices to form a module where the mult method of the second hyperslice replaces the abstract mult method of the first hyperslice. We model such a composition by defining the following three pieces of advice, using $\boldsymbol{M}_G$:

$$(\textsc{Val} \wedge \mathsf{S} = \mathsf{MultPoint}) \rhd$$
$$\varsigma() \quad [\mathsf{slice1} = \varsigma(\mathsf{s})\ \mathsf{PointSlice},\ \mathsf{slice2} = \varsigma(\mathsf{s})\ \mathsf{MultSlice},\ \mathsf{n} = \varsigma(\mathsf{s})\ (\mathsf{s.slice1.outer} \Leftarrow \mathsf{s}).\mathsf{n},\ \mathsf{pos} = \varsigma(\mathsf{s})\ (\mathsf{s.slice1.outer} \Leftarrow \mathsf{s}).\mathsf{pos},\ \mathsf{mult} = \varsigma(\mathsf{s})\ (\mathsf{s.slice2.outer} \Leftarrow \mathsf{s}).\mathsf{mult}]$$

$$\textsc{Upd} \wedge \mathsf{S} = \{\mathsf{slice1},\mathsf{slice2},\mathsf{n},\mathsf{pos},\mathsf{mult}\} \wedge (\mathsf{k} = \mathsf{n} \vee \mathsf{k} = \mathsf{pos}) \rhd$$
$$\varsigma(\mathsf{t},\mathsf{r}) \quad [\mathsf{slice1} = \varsigma(\mathsf{s})\ \mathsf{proceed}_{\textsc{Upd}}(\mathsf{t.slice1},\varsigma(\mathsf{t})\mathsf{r}),\ \mathsf{slice2} = \varsigma(\mathsf{s})\ \mathsf{t.slice2},\ \mathsf{n} = \varsigma(\mathsf{s})\ (\mathsf{s.slice1.outer} \Leftarrow \mathsf{s}).\mathsf{n},\ \mathsf{pos} = \varsigma(\mathsf{s})\ (\mathsf{s.slice1.outer} \Leftarrow \mathsf{s}).\mathsf{pos},\ \mathsf{mult} = \varsigma(\mathsf{s})\ (\mathsf{s.slice2.outer} \Leftarrow \mathsf{s}).\mathsf{mult}]$$

$$\textsc{Upd} \wedge \mathsf{S} = \{\mathsf{slice1},\mathsf{slice2},\mathsf{n},\mathsf{pos},\mathsf{mult}\} \wedge \mathsf{k} = \mathsf{mult} \rhd$$
$$\varsigma(\mathsf{t},\mathsf{r}) \quad [\mathsf{slice1} = \varsigma(\mathsf{s})\ \mathsf{t.slice1},\ \mathsf{slice2} = \varsigma(\mathsf{s})\ \mathsf{proceed}_{\textsc{Upd}}(\mathsf{t.slice2},\ \varsigma(\mathsf{t})\mathsf{r}),\ \mathsf{n} = \varsigma(\mathsf{s})\ (\mathsf{s.slice1.outer} \Leftarrow \mathsf{s}).\mathsf{n},\ \mathsf{pos} = \varsigma(\mathsf{s})\ (\mathsf{s.slice1.outer} \Leftarrow \mathsf{s}).\mathsf{pos},\ \mathsf{mult} = \varsigma(\mathsf{s})\ (\mathsf{s.slice2.outer} \Leftarrow \mathsf{s}).\mathsf{mult}]$$

The first advice causes the basic constant MultPoint in expressions to be replaced with an object that is comprised of the two hyperslices, plus methods to direct selection on n and pos to the first hyperslice, and selection on mult to the second hyperslice, after first setting the outer field of the slice appropriately. The two pieces of update advice are identical in design to that used for open classes in Section 3.1.2.

With this advice, we can show results like [6]:

$$\epsilon \vdash ((\mathsf{MultPoint.n} \Leftarrow \varsigma(\mathsf{y})4).\mathsf{mult} \Leftarrow \varsigma(\mathsf{y})3).\mathsf{pos} \rightsquigarrow 12$$

[5] Invoking mult on a PointSlice without providing a non-abstract implementation of mult would result in an infinite regress.

We directly model HyperJ using $\varsigma_{asp}(\boldsymbol{M}_G)$. The use of advice on basic constants models the static nature of HyperJ's compositions. The advice representing compositions could be generated by a context-free translation; all that is needed is a mapping indicating from which slice each member of the composed object comes. It is straightforward to define advice to model more complicated compositions.

3.3 Adaptive Methods

In adaptive methods, a *traversal strategy* abstractly specifies a walk of the object graph of a program [11]. Reflection is used to reify this abstract strategy into an actual walk of the graph at run-time. To model adaptive methods with ς_{asp} we need a point cut description language with reflective capabilities; specifically, we need to be able to find the set of labels of an object o, and invoke those labels that return the sub-objects of o. We do this by establishing a convention that labels for fields begin with "f_" and defining a point cut description language, $\boldsymbol{M}_R = \langle \boldsymbol{\mathcal{C}}_R, match_R \rangle$, that extends $\boldsymbol{M}_G$ with a mechanism to quantify over the fields of an object [6]. We sketch the mechanism here.

All point cut descriptions in $\boldsymbol{\mathcal{C}}_G$ (see Figure 5) are valid in $\boldsymbol{\mathcal{C}}_R$. Additionally the suffix "$\cdot \forall\, l \in \mathsf{fieldsOf(S)}$" may be added to any of $\boldsymbol{\mathcal{C}}_G$'s point cut descriptions. This suffix causes $match_R$ to create a sequence of advice from a single matching piece of advice. The generated sequence has one element for each field in the target object of the join point.

For a point cut description without the quantifier suffix, $match_R$ is identical to $match_G$. For a point cut description $pcd_G \cdot \forall\, l \in \mathsf{fieldsOf(S)}$, $match_R$ is defined as follows:

$$match_R(pcd_G \cdot \forall\, l \in \mathsf{fieldsOf(S)} \rhd \varsigma(\overrightarrow{y})b, \langle \rho, \mathcal{K}, S, k \rangle) =$$
$$\begin{cases} \bullet & \text{if } match_G(pcd_G \rhd \varsigma(y)b, \langle \rho, \mathcal{K}, S, k \rangle) = \bullet \\ \langle \varsigma(\overrightarrow{y})b_1, \ldots, \varsigma(\overrightarrow{y})b_m \rangle & \text{otherwise} \end{cases}$$

where $\{l_1, \ldots, l_m\}$ is the set of field labels in S and each b_i is formed from b by first finding all occurrences of l as a selection or update label, and then replacing them with l_i.

For example, given the advice:

$$\textsc{Ivk} \wedge \mathsf{k} = \mathsf{print} \cdot \forall\, \mathsf{field} \in \mathsf{fieldsOf(S)} \rhd$$
$$\quad \varsigma(\mathsf{x})\ \mathsf{proceed}_{\textsc{Ivk}}(\mathsf{x.field.print})$$

and the join point $\langle \textsc{Ivk}, \epsilon, \{\mathsf{f\_left}, \mathsf{f\_right}, \mathsf{sum}\}, \mathsf{print} \rangle$, $match_R$ will return the sequence:

$$\langle\ \varsigma(\mathsf{x})\ \mathsf{proceed}_{\textsc{Ivk}}(\mathsf{x.f\_left.print}),$$
$$\quad \varsigma(\mathsf{x})\ \mathsf{proceed}_{\textsc{Ivk}}(\mathsf{x.f\_right.print})\ \rangle$$

We model traversals by using update advice to walk the object graph. Update advice has two parameters. We use one parameter to track the root of the object (sub-)graph to be traversed, using $\varsigma_{asp}(\boldsymbol{M}_R)$'s reflection to follow the traversal strategy. We use the other parameter to hold a visitor object for accumulating results. Details are available in the companion technical report [6].

4. DISCUSSION AND FUTURE WORK

Filman and Friedman argue that the two essential features of an aspect-oriented language are quantification and obliviousness [7]. *Quantification* is the ability to specify that a block of code is to be executed at multiple points in the body of a program. *Obliviousness* is the property that the base program need not explicitly mention aspect-oriented code in order for such code to be executed.

Our calculus directly models quantification and obliviousness. Quantification is modeled by allowing single pieces of advice to bind to multiple join points. Obliviousness is modeled by not requiring base program terms to explicitly mention aspect-oriented code. Our model for AspectJ's point cuts, like the language itself, allows the base program to be completely oblivious to the aspect-oriented code. Our model for HyperJ, like the language itself, allows a hyperslice to be oblivious to others' implementations, but requires non-oblivious composition and client code. Similarly, our model for adaptive methods requires the base program to invoke the traversal, but allows it to be oblivious to the actual object graph structure. The correspondence in obliviousness properties between our models and the actual languages supports our contention that $\varsigma_{asp}(\boldsymbol{M}_G)$ directly models these languages.

Some have argued that obliviousness means that a base program must not reference aspect-oriented code and that one should be able to reason about the base program without considering the aspect-oriented code.[6] We take exception to this second claim and have previously argued, albeit informally, that it may be necessary to sacrifice obliviousness in order to modularly reason about aspect-oriented programs [4, 18]. We plan to formally study the reasoning properties of aspect-oriented languages in an attempt to understand the relationship between obliviousness and modular reasoning. Because ς_{asp} is explicitly aspect-oriented, directly modeling both quantification and obliviousness, it will serve well for our planned study.

5. RELATED WORK

Masuhara and Kiczales also examine models for HyperJ, adaptive methods, and AspectJ's join points and open classes [12]. They develop a series of interpreters, written in Scheme, for subsets of each aspect-oriented language and compare and contrast those interpreters. However, because they abstract away fewer features of the languages, and because their interpreters are Scheme programs, their approach is less amenable to formal study, compared to ς_{asp}.

Walker, Zdancewic, and Ligatti present a core aspect-oriented calculus based on the simply-typed lambda calculus [17]. However, their calculus is completely unoblivious—sub-terms must be explicitly labeled in order for advice to be applied to those terms. The paper demonstrates how an aspect-oriented variant of ML, MinAML, which is oblivious, may be translated into the core calculus. The lack of any obliviousness in MinAML makes it less suitable than ς_{asp} for studying the impact of obliviousness properties on reasoning.

Walker, Zdancewic, and Ligatti [17, §3.2] also extend their core calculus with objects à la Abadi and Cardelli. However, the resulting calculus is significantly more complex than ς_{asp}, as it includes the simply-typed λ calculus, the ς calculus, and the label system for attaching advice. It is not clear that their calculus can be used to model open classes, HyperJ, or adaptive methods, because it lacks the update advice we rely on to model of these languages.

[6] We have not seen published arguments for this position, but we have heard it expressed at Foundations of Aspect-Oriented Languages 2002 and 2003, and in conversations.

Jagadeesan, Jeffrey, and Reily present an aspect-oriented calculus that is class-based and models multi-threaded programs [8]. They examine equivalence between aspect-oriented programs in the calculus and the same programs translated into an object-oriented subset of the calculus. This equivalence demonstrates the correctness of their translation algorithm. Although well-suited to this purpose, their calculus is overly complex for our planned studies of reasoning properties. Their calculus only considers point cut descriptions that are boolean combinations of call and execution join points.

Other formal semantics have been described with various join point models. Wand, Kiczales, and Dutchyn give a denotational semantics for a first-order procedural language with join points for procedure call and procedure and advice execution [19]. Lämmel gives an operational semantics for an imperative object oriented language with join points for method calls [10]. These languages only model a small fraction of AspectJ's join points, and, unlike ς_{asp}, they do not model open classes, HyperJ, or adaptive methods.

6. CONCLUSION

In this paper we have described the parameterized aspect calculus, $\varsigma_{asp}(\boldsymbol{M})$, for modeling aspect-oriented languages. We have demonstrated how $\varsigma_{asp}(\boldsymbol{M})$ can be used to directly model several such existing languages including key features of AspectJ, HyperJ, and adaptive methods. These models encompass a wide variety of aspect-oriented features, while maintaining the quantification and oblviousness properties of the original language. Because of the directness of its modeling and the preservation of key properties, $\varsigma_{asp}(\boldsymbol{M})$ provides an excellent basis for studying the formal properties of aspect-oriented languages.

7. ACKNOWLEDGMENTS

The work of Clifton and Leavens was supported in part by the US National Science Foundation under grants CCR-0097907 and CCR-0113181.

8. REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, NY, 1996.

[2] M. Akşit, editor. *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*. ACM Press, Mar. 2003.

[3] AspectJ Team. The AspectJ programming guide. Available from `http://eclipse.org/aspectj`, Oct. 2003.

[4] C. Clifton and G. T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. Technical Report 03-01a, Iowa State University, Department of Computer Science, Mar. 2003.

[5] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, New York, Oct. 2000. ACM.

[6] C. Clifton, G. T. Leavens, and M. Wand. Formal definition of the parameterized aspect calculus. Technical Report 03-12b, Iowa State University, Department of Computer Science, Nov. 2003.

[7] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In M. Akşit, S. Clarke, T. Elrad, and R. E. Filman, editors, *Aspect-Oriented Software Development*. Addison-Wesley, Reading, MA, to appear.

[8] R. Jagadeesan, A. Jeffrey, and J. Reily. A calculus of untyped aspect-oriented programs. In L. Cardelli, editor, *ECOOP 2003, European Conference on Object-Oriented Programming, Darmstadt, Germany*, volume 2743 of *Lecture Notes in Computer Science*, pages 54–73. Springer-Verlag, New York, NY, 2003.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Berlin, June 2001.

[10] R. Lämmel. A semantical approach to method-call interception. In G. Kiczales, editor, *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 41–55. ACM Press, Apr. 2002.

[11] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44(10):39–41, Oct. 2001.

[12] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP 2003–Object-Oriented Programming 17th European Conference*, pages 2–28. Springer-Verlag, July 2003. Lecture Notes in Computer Science, Volume 2743.

[13] T. Millstein and C. Chambers. Modular statically typed multimethods. In R. Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628 of *Lecture Notes in Computer Science*, pages 279–303. Springer-Verlag, New York, NY, June 1999.

[14] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Commun. ACM*, 44(10):43–50, Oct. 2001.

[15] D. Sereni and O. de Moor. Static analysis of aspects. In Akşit [2], pages 30–39.

[16] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In Akşit [2], pages 158–167.

[17] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*, pages 127–139. ACM Press, Aug. 2003.

[18] M. Wand. Understanding aspects: extended abstract. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 299–300. ACM Press, 2003.

[19] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Prog. Lang. Syst.*, 2003. to appear.