A Calculus of Untyped Aspect-Oriented Programs

Radha Jagadeesan, Alan Jeffrey, and James Riely

CTI, DePaul University

Abstract. Aspects have emerged as a powerful tool in the design and development of systems, allowing for the encapsulation of program transformations. The dynamic semantics of aspects is typically specified by appealing to an underlying object-oriented language via a compiler transformation known as *weaving*. This treatment is unsatisfactory for several reasons. Firstly, this semantics violates basic modularity principles of object-oriented programming. Secondly, the converse translation from object-oriented programs into an aspect language has a simple canonical flavor. Taken together, these observations suggest that aspects are worthy of study as primitive computational abstractions in their own right. In this paper, we describe an aspect calculus and its operational semantics. The calculus is rich enough to encompass many of the features of extant aspect-oriented frameworks that do not involve reflection. The independent description of the dynamic semantics of aspects enables us to specify the correctness of a weaving algorithm. We formalize weaving as a translation from the aspect calculus to a class-based object calculus, and prove its soundness.

1 Introduction

In this paper we give the dynamic semantics for an aspect-based language and prove the correctness of weaving with respect to that semantics.

Aspects: A Short Introduction. Aspects have emerged as a powerful tool in the design and development of systems [4, 14, 19, 16, 15, 2]. We begin with a short example to introduce the basic vocabulary of aspect-oriented programming and illustrate the underlying issues. Although our examples throughout the paper are couched in terms of AspectJ (`http://www.aspectj.org`), our study is more general in scope.

Suppose that `L` is a class realizing a useful library. Suppose further say that we are interested in timing information about a method `foo()` in `L`. The following AspectJ code addresses this situation. It is noteworthy and indicative of the power of the aspect framework that

- the profiling code is localized in the following aspect,
- the existing client and library source code is left untouched, and
- the responsibility for profiling all `foo()` calls resides with the AspectJ compiler.

```
aspect TimingMethodInvocation {
   Timer timer = new Timer();
   void around(): call (void L.foo()) {
      timer.start();  proceed();  timer.stop;
      System.out.println(timer.getTime());
   }}
```

This aspect is intended to trap all invocations to `foo()` in `L`. An aspect may *advise* methods, causing additional code to be executed whenever a method of interest is called. The set of interesting methods is specified using a *pointcut*, here `call (void L.foo())`. The advice itself is a sequence of commands. The example uses `around` advice. The intended execution semantics is as follows: a call to `foo()` invokes the code associated with the advice; in the example, the timer is started. The underlying `foo()` method is invoked when control reaches `proceed()`. Upon termination of `foo()`, control returns to the advice; in the example, the timer is stopped and the elapsed time displayed on the screen.

In many aspect-based languages, the intended execution semantics is realized by a compile-time process called *weaving*. Because the advice is attached to a *call pointcut* in the example, the weaving algorithm replaces each call to `foo()` with a call to the advice; it alters the client code, leaving the library `L` untouched. In this light, it is not surprising that dispatch of call pointcuts is based on the *compile-time type* of the receiver of `foo()`.

The converse effect is achieved using an *execution pointcut*. Our example can be altered to use execution pointcuts by replacing `call (void L.foo())` with `execution (void L.foo())`. In this case the weaving algorithm alters the library, leaving the client untouched. Dispatch of execution pointcuts is based on the *runtime type* of the receiver of `foo()`.

In general, there may be several pieces of advice attached to a method, and therefore there must be an ordering on advice which determines the order of execution. In AspectJ, for example, the textual order of declarations is used.

Advice may also take parameters, and these parameters may be passed on to the next piece of advice via `proceed`. In particular, in both call and execution advice one can define a binder for `target`, the object receiving the message. In call advice, one can additionally bind `this`, the object sending the message.

Aspects interfere with OO reasoning. Much of the power in aspect-oriented programming lies in the ability to intercept method calls. This power, however, comes at the price of breaking object-oriented encapsulation and the reasoning that it allows. As a simple example, consider the following declarations:

```
class C { void foo(){..} }
class D extends C { }
```

In an object-oriented language, this definition is indistinguishable from the following:

```
class C { void foo(){..} }
class D extends C { void foo(){ super.foo(); } }
```

The following aspect, however, distinguishes them:

```
aspect SpotInheritance {
   void around(): execution (void D.foo()) {
      System.out.println("aspect in action");
   }}
```

In the first declaration, the execution advice cannot attach itself to `foo` in `D`, since `D` does not declare the method; it inherits it. It cannot attach itself either to `C`, since the advice is intended for `D` alone. The effect of the aspect is seen only in the second declaration where `foo()` is redeclared, albeit trivially.

Unfortunately, interference with object-oriented reasoning does not stop there. As a second example, consider that behavioral changes caused by aspects need not be inherited down the class hierarchy. The following aspect distinguishes objects of type `C` that are not also of type `D`:

```
aspect OnlySuperclass {
   void around(): (void execution(C.foo()))
              && !(void execution(D.foo())) {
      System.out.println("aspect in action");
   }}
```

These examples indicate that one cannot naively extend object-oriented reasoning to aspect-oriented programs.

Our aims: Reductionism and a specification for weaving. Our approach to understanding aspect-oriented programming is based on an aspect calculus. We have attempted to define the essential features of an aspect-oriented language, leaving many pragmatic programming constructs out. To begin with, we do not include the `aspect` container in our language, taking advice to be primitive. In addition, we study only `around` advice. Some other forms of advice can be derived. For example, AspectJ includes *before advice*, which executes just before a method is called; "`before(){C;}`" can be encoded as "`around(){C; proceed();}`". We also define a simple logic to describe pointcuts, built up from the call and execution primitives described above. Aspect-oriented languages such as AspectJ have a rich collection of pointcuts, including ones that rely on reflection. While many forms of pointcuts can be encoded in our language, we do not address reflection. In this paper, we also focus strictly on dynamics, avoiding issues related to the static semantics and type checking.

Perhaps more important than providing a core calculus, the source-level semantics for aspects provides a specification for the weaving algorithm. Rather than using transformation to define the semantics, we are able to prove the *correctness* of the weaving transformation; we prove that woven programs (where all aspects have been removed) perform computation exactly as specified by the original aspect program.

In one respect, our aspect calculus is richer than statically woven languages such as AspectJ, allowing for the the dynamic addition of advice to a running program. Clearly, programs that dynamically load advice affecting existing classes cannot be woven statically. We define a notion of *weavability*, which excludes such programs, and prove the correctness of weaving with respect to weavable programs only.

The rest of this paper. We define a class-based language in Section 2. The class-based language is the foundation for the aspect-based language introduced in Section 3. In Section 4 we describe the weaving algorithm, which translates programs in the aspect-based languages into the class-based language. Section 5 states the correctness theorem; all proofs can be found in the full version of the paper. We conclude with a survey of related work.

2 A Class-based Language

In this section, we describe an untyped class-based language. In contrast to untyped object calculi, our language includes a primitive notion of *class*. This approach simplifies the later discussion of aspects, whose advice is bound to classes, rather than objects.

Previous work on class-based languages has concentrated on translations from class-based languages into polymorphic λ-calculi [5] or to object-based languages such as the ς-calculus [1]. For example, this is the technique used in giving the semantics of LOOM [6], PolyTOIL [7] and Moby [11]. There is less literature on providing a direct semantics for class-based languages; notable exceptions are Featherweight Java [12] and Java$_s$ [9]. Our semantics is heavily influenced by Featherweight Java, but is for a multi-threaded language of mutable objects rather than a single-threaded language of immutable objects; in addition, we do not address issues of genericity or of translating away inner classes [13].

NOTATION. For any metavariable x, we write $\vec{x}$ for ordered sequences of x's, and $\bar{x}$ for unordered sequences of x's.

A program P has the form $(\bar{D} \vdash \bar{H})$, where $\bar{D}$ is a set of declarations and $\bar{H}$ is a set of heap allocated threads and objects. A class declaration "$\mathsf{class}\ c <: d\ \{ \ldots m(\vec{x})\ \{\vec{C}\} \ldots \}$" must indicate the superclass d and a set of method declarations. Fields are not declared since the language is untyped. The superclass relation is terminated in the undeclared class "Object". An object declaration, "$\mathsf{obj}\ o : c\ \{ \ldots f = v \ldots \}$" must indicate the actual class of the object and the values of the fields. A thread declaration, "$\mathsf{thrd}\ p\{S\}$" names a controlling object p and a stack S, which contains a sequence of commands to perform. If a thread is executing a method on behalf of object p, then p will be the controlling object. We include the controlling object only for compatibility with the aspect-based language; it is not used here.

The dynamic semantics of the class-based language is described as a transformation of programs:

$$P \rightarrow P'$$

Let us consider a few examples. As a simple example of a command, the values of fields in objects can be retrieved by dereferencing the heap.

$$\begin{array}{l}\mathsf{obj}\ o : c\ \{ \ldots f = v \ldots \} \\ \mathsf{thrd}\ p\{\mathsf{let}\ x = o.f;\ \}\end{array} \rightarrow \begin{array}{l}\mathsf{obj}\ o : c\ \{ \ldots f = v \ldots \} \\ \mathsf{thrd}\ p\{\mathsf{let}\ x = v;\ \}\end{array}$$

Symmetrically, a field in an object can be set to store a new value.

$$\begin{array}{l}\mathsf{obj}\ o : c\ \{ \ldots f = u \ldots \} \\ \mathsf{thrd}\ p\{\mathsf{set}\ o.f = v;\ \}\end{array} \rightarrow \begin{array}{l}\mathsf{obj}\ o : c\ \{ \ldots f = v \ldots \} \\ \mathsf{thrd}\ p\{\}\end{array}$$

Threads may include "nested" class declarations that are loaded dynamically:

$$\mathsf{thrd}\ p\{\mathsf{new\ class}\ c <: d\ \{ \ldots \};\ \} \rightarrow \begin{array}{l}\mathsf{class}\ c <: d\ \{ \ldots \} \\ \mathsf{thrd}\ p\{\ \}\end{array}$$

Table 1 Class-Based Syntax

$a,..,z$	*Name*	$C,B ::=$	*Command*
$P,Q ::= (\bar{D} \vdash \bar{H})$	*Program*	$\mathsf{new}\ \bar{D};$	New Class
		$\mathsf{new}\ \bar{H};$	New Heap Element
$D,E ::= \mathsf{class}\ c <: d\ \{\bar{M}\}$	*Declaration*	$\mathsf{return}\ v;$	Return
$M ::= m(\vec{x})\ \{\vec{C}\}$	*Method*	$\mathsf{let}\ x = v;$	Value
$H,G ::=$	*Heap Element*	$\mathsf{let}\ x = o.m(\vec{v});$	Dynamic Message
$\mathsf{obj}\ o : c\ \{\bar{F}\}$	Object	$\mathsf{let}\ x = o.c :: m(\vec{v});$	Static Message
$\mathsf{thrd}\ o\{S\}$	Thread	$\mathsf{let}\ x = o.f;$	Get Field
$F ::= f = v$	*Field*	$\mathsf{set}\ o.f = v;$	Set Field
$S,T ::=$	*Call Stack*		
$\vec{C}$	Current Frame		
$\mathsf{let}\ x = o\{S\}; \vec{C}$	Pushed Frame		

The most important reduction rules, however, are those involving method invocation. In a dynamically dispatched message, we first look up the dynamic type of the object. Next, we move up the superclass chain till we find a class where the method is actually defined. Finally, once we have discovered the class where the method is defined, reduction proceeds via a standard substitution of parameters for the method, instantiating of this with the actual receiver of the method.

$$\begin{array}{l}\mathsf{class\ d <: Object}\ \{\ldots\ \mathsf{m(x)}\ \{\vec{B}\}\ldots\}\\ \mathsf{class\ c <: d}\ \{\ldots\}\\ \mathsf{obj\ o : c}\ \{\ldots\}\\ \mathsf{thrd\ p}\{\mathsf{o.m(v)\ ;}\}\end{array} \rightarrow \begin{array}{l}\mathsf{class\ d <: Object}\ \{\ldots\ \mathsf{m(x)}\ \{\vec{B}\}\ldots\}\\ \mathsf{class\ c <: d}\ \{\ldots\}\\ \mathsf{obj\ o : c}\ \{\ldots\}\\ \mathsf{thrd\ p}\{\vec{B}[\mathsf{o}/\mathsf{this}, \mathsf{v}/\mathsf{x}]\}\end{array}$$

We include statically dispatched messages to encode superclass calls. In class c which extends d, "$\mathsf{super}.m(\vec{v})\ ;$" is encoded "$\mathsf{this}.d :: m(\vec{v})\ ;$".

2.1 Syntax

The syntax is given in Table 1. In definitions and examples, we write "_" to stand for an element of any syntactic category that is not of interest.

Lower-case letters a–z range over a set of names. "Object", "this", "target" and "proceed" are reserved names. Although all names are drawn from a single set, our use of names is disciplined to improve readability. We use c–e for class names; f for field names; m for method names; o–q for object reference names; x–z for variables; v for values (object references or variables); a–b for advice. Advice is discussed in the following section, where we will also assume a fixed total order on names, $n \prec m$. We may write a collection of names as "$a, \bar{a}$" to indicate that a is ordered before any of the names in $\bar{a}$.

Table 2 Reduction

$$\frac{\bar{D} \ni \mathsf{class}\ c <: \_\ \{\bar{M}, m(\vec{x})\ \{\vec{C}\}\}}{\bar{D} \vdash \mathrm{body}(c::m) = (\vec{x})\vec{C}}\ (\mathrm{L_C\text{-}THIS})$$

$$\frac{\bar{D} \ni \mathsf{class}\ c <: d\ \{\bar{M}\} \quad \bar{M} \not\ni m(\_)\ \{\_\} \quad \bar{D} \vdash \mathrm{body}(d::m) = (\vec{x})\vec{C}}{\bar{D} \vdash \mathrm{body}(c::m) = (\vec{x})\vec{C}}\ (\mathrm{L_C\text{-}SUPER})$$

$$\frac{(\bar{D} \vdash \bar{H}, \mathsf{thrd}\ q\{S\}) \rightarrow (\bar{D}' \vdash \bar{H}', \mathsf{thrd}\ q\{S'\})}{(\bar{D} \vdash \bar{H}, \mathsf{thrd}\ p\{\mathsf{let}\ x = q\{S\}; \vec{C}\}) \rightarrow (\bar{D}' \vdash \bar{H}', \mathsf{thrd}\ p\{\mathsf{let}\ x = q\{S'\}; \vec{C}\})}\ (\mathrm{R_C\text{-}LET})$$

$$\frac{}{(\bar{D} \vdash \bar{H}, \mathsf{thrd}\ p\{\mathsf{let}\ x = v; \vec{C}\}) \rightarrow (\bar{D} \vdash \bar{H}, \mathsf{thrd}\ p\{\vec{C}[v/x]\})}\ (\mathrm{R_C\text{-}VALUE})$$

$$\frac{}{(\bar{D} \vdash \bar{H}, \mathsf{thrd}\ p\{\mathsf{let}\ x = q\{\mathsf{return}\ v; \vec{B}\}; \vec{C}\}) \rightarrow (\bar{D} \vdash \bar{H}, \mathsf{thrd}\ p\{\mathsf{let}\ x = v; \vec{C}\})}\ (\mathrm{R_C\text{-}RETURN})$$

$$\frac{}{(\bar{D} \vdash \bar{H}, \mathsf{thrd}\ p\{\mathsf{return}\ v; \vec{C}\}) \rightarrow (\bar{D} \vdash H)}\ (\mathrm{R_C\text{-}GARBAGE})$$

$$\frac{\bar{H} \ni \mathsf{obj}\ o : c\ \{\_\} \quad \bar{D} \vdash \mathrm{body}(c::m) = (\vec{x})\vec{B}}{(\bar{D} \vdash \bar{H}, \mathsf{thrd}\ p\{\mathsf{let}\ x = o.m(\vec{v}); \vec{C}\}) \rightarrow (\bar{D} \vdash \bar{H}, \mathsf{thrd}\ p\{\mathsf{let}\ x = o\{\vec{B}[o/\mathsf{this}, \vec{v}/\vec{x}]\}; \vec{C}\})}\ (\mathrm{R_C\text{-}DYN\text{-}MSG})$$

$$\frac{\text{domains of } \bar{D} \text{ and } \bar{E} \text{ are disjoint}}{(\bar{D} \vdash \bar{H}, \mathsf{thrd}\ p\{\mathsf{new}\ \bar{E}; \vec{C}\}) \rightarrow (\bar{D}, \bar{E} \vdash \bar{H}, \mathsf{thrd}\ p\{\vec{C}\})}\ (\mathrm{R_C\text{-}DEC})$$

$$\frac{\bar{D} \vdash \mathrm{body}(c::m) = (\vec{x})\vec{B}}{(\bar{D} \vdash \bar{H}, \mathsf{thrd}\ p\{\mathsf{let}\ x = o.c::m(\vec{v}); \vec{C}\}) \rightarrow (\bar{D} \vdash \bar{H}, \mathsf{thrd}\ p\{\mathsf{let}\ x = o\{\vec{B}[o/\mathsf{this}, \vec{v}/\vec{x}]\}; \vec{C}\})}\ (\mathrm{R_C\text{-}STC\text{-}MSG})$$

$$\frac{\text{domains of } \bar{H} \text{ and } \bar{G} \text{ are disjoint}}{(\bar{D} \vdash \bar{H}, \mathsf{thrd}\ p\{\mathsf{new}\ \bar{G}; \vec{C}\}) \rightarrow (\bar{D} \vdash \bar{H}, \bar{G}, \mathsf{thrd}\ p\{\vec{C}\})}\ (\mathrm{R_C\text{-}HEAP})$$

$$\frac{\bar{H} \ni \mathsf{obj}\ o : c\ \{\bar{F}, f = v\}}{(\bar{D} \vdash \bar{H}, \mathsf{thrd}\ p\{\mathsf{let}\ x = o.f; \vec{C}\}) \rightarrow (\bar{D} \vdash \bar{H}, \mathsf{thrd}\ p\{\mathsf{let}\ x = v; \vec{C}\})}\ (\mathrm{R_C\text{-}GET})$$

$$\frac{}{(\bar{D} \vdash \bar{H}, \mathsf{obj}\ o : c\ \{\bar{F}, f = u\}, \mathsf{thrd}\ p\{\mathsf{set}\ o.f = v; \vec{C}\}) \rightarrow (\bar{D} \vdash \bar{H}, \mathsf{obj}\ o : c\ \{\bar{F}, f = v\}, \mathsf{thrd}\ p\{\vec{C}\})}\ (\mathrm{R_C\text{-}SET})$$

We define the notion of *bound name* for method declarations and command sequences. The method declaration "$m(\vec{x})\ \{\vec{C}\}$" binds $\vec{x}$ and this, the scope is $\vec{C}$. The class declaration "new class $c <: d\ \{\bar{M}\}; \vec{C}$" binds c, with scope $\bar{M}$ and $\vec{C}$. The object declaration "new obj $o : c\ \{\bar{F}\}; \vec{C}$" binds o, with scope $\bar{F}$ and $\vec{C}$. Each let-command sequence "let $x = \ldots; \vec{C}$", binds x, with scope $\vec{C}$. Command sequences associate to the right, so "$C_1\ C_2\ C_3$" should be read "$C_1(C_2\ C_3)$"; the scope of variables bound in C_1 includes C_2 and C_3. Note that there are no binders for method or field names; the usual semantics requires a static typing system, which we purposefully avoid here. We identify programs up to renaming of bound names and define substitution $\vec{C}[v/x]$ as usual.

2.2 Dynamic semantics

Computation proceeds by executing the command sequences contained in threads. Commands may include declaration of classes "$\mathsf{new}\ \overline{D}\,;$" or heap elements "$\mathsf{new}\ \overline{H}\,;$". The value stored in an object field can be retrieved "$\mathsf{let}\ x = o.f\,;$" and set "$\mathsf{set}\ o.f = v\,;$". Method calls may be dispatched using the dynamic type of the object "$\mathsf{let}\ x = o.m(\vec{v})\,;$" or a statically chosen type "$\mathsf{let}\ x = o.c :: m(\vec{v})\,;$".

A pushed frame "$\mathsf{let}\ x = p\{S\}\,;\vec{C}$" successfully terminates in a return command which removes the remainder of S, leaving $\vec{C}$ to execute; "$\mathsf{let}\ x = p\{\mathsf{return}\ v\,;\vec{B}\}\,;\vec{C}$" reduces to "$\mathsf{let}\ x = v\,;\vec{C}$", which is then further reduced via substitution to "$\vec{C}[v/x]$". A top level return "$\mathsf{thrd}\ p\{\mathsf{return}\ v\,;\vec{C}\}$" causes the thread to be garbage collected.

The reduction rules $P \rightarrow P'$ are given in Table 2. The rules ($\mathrm{R_C}$-LET), ($\mathrm{R_C}$-RETURN) and ($\mathrm{R_C}$-GARBAGE) deal with pushed frames. The rule ($\mathrm{R_C}$-VALUE) allows returned values to be substituted through for the variables to which they are bound. The rules ($\mathrm{R_C}$-DEC) and ($\mathrm{R_C}$-HEAP) allow threads to create new classes, objects and threads. These rules require alpha-renaming to make the domains disjoint, allowing generation of new class, object and thread names. The rules ($\mathrm{R_C}$-GET) and ($\mathrm{R_C}$-SET) allow for the manipulation of fields.

The rules ($\mathrm{R_C}$-DYN-MSG) and ($\mathrm{R_C}$-STC-MSG) perform beta reduction on method calls; in the dynamic case, the method is determined by the actual class of the object o; in the static case, the method is determined by the annotated method call $c :: m$. These rules use an auxiliary relation for method lookup "$\overline{D} \vdash \mathrm{body}(c :: m) = (\vec{x})\vec{C}$", also defined in Table 2. The rule ($\mathrm{L_C}$-THIS) allows for a method body to be retrieved from the class which declares it, whereas ($\mathrm{L_C}$-SUPER) specifies that if a method is not declared in a class, then the superclass should be checked. Note that *body* defines a partial function.

3 An Aspect-based Language

The move from a class-based language to an aspect-based language involves three new pieces of syntax: aspect declarations, advised method calls and proceed calls.

An aspect declaration, "$\mathsf{adv}\ a(\vec{x}) : \phi\ \{\vec{C}\}$" has three essential components. The name a allows references to the aspect from elsewhere in the program. The command sequence $\vec{C}$ species *what* to execute and the pointcut ϕ specifies *when*. A pointcut specifies the set of methods that are affected by this aspect; formally pointcuts are presented as elements of the boolean algebra whose atoms are execution pointcuts, $\mathsf{exec}(c :: m)$, and call pointcuts, $\mathsf{call}(c :: m)$.

An advised method call "$\mathsf{let}\ x = o.m\,[\overline{a}\,;\overline{b}]\,(\vec{v})\,;$" specifies a set $\overline{a}$ of call advice and a set $\overline{b}$ of execution advice. For simplicity, we assume that there is a fixed total ordering on the names of advice ($n \prec m$) which determines the execution order; we do not allow declarations of advice precedence. The advice sets $\overline{a}$ and $\overline{b}$ determine the semantics of advised method calls; the method name m is an annotation required only to define the weaving function for call advice in the Section 4.

The final new command is "$\mathsf{let}\ x = \mathsf{proceed}\,(\vec{v})\,;$" which is intended to appear in the body of advice. This command plays a crucial role in the operational semantics as sketched below.

Due to the presence of call advice, we must know the static (declared) type of an object reference, in addition to its dynamic (actual) type. Thus, in the aspect language, each dynamically dispatched method call "$\mathsf{let}\ x = o : c.m(\vec{v})\ ;$" must be annotated with a static type c. We follow AspectJ in ignoring call advice for super calls, here modeled by statically dispatched messages. We could easily adapt our semantics to execute call advice on static messages as well.

In the aspect language, classes have the form "$\mathsf{class}\ c <: d\ \{ \ldots m[\overline{a}\,;\overline{b}] \ldots \}$". There are no commands directly associated with classes, rather, they are indirectly associated using the advice sets $\overline{a}$ and $\overline{b}$. Method bodies in class declarations would be redundant, as demonstrated below.

Let us first consider a few examples. Given a method call in the aspect-oriented calculus, we first lookup the call and execution advice associated with the method to build an advised method call. Call advice lookup is based on the declared type, whereas execution advice lookup is based on the actual type.

$$\begin{array}{l} \mathsf{adv\ a(\ldots) : call(c :: m)\ \{\ldots\}} \\ \mathsf{adv\ b(\ldots) : exec(d :: m)\ \{\ldots\}} \\ \mathsf{obj\ o : d\ \{\ldots\}} \\ \mathsf{thrd\ p\{o : c.m(v)\ ;\}} \end{array} \quad \rightharpoonup \quad \begin{array}{l} \mathsf{adv\ a(\ldots) : call(c :: m)\ \{\ldots\}} \\ \mathsf{adv\ b(\ldots) : exec(d :: m)\ \{\ldots\}} \\ \mathsf{obj\ o : d\ \{\ldots\}} \\ \mathsf{thrd\ p\{o.m[a\ ;\ b]\ (v)\ ;\}} \end{array}$$

Now, consider an advised message where the call advice list is nonempty:

$$\begin{array}{l} \mathsf{adv\ a(x) :\ldots\ \{}\vec{C}\mathsf{\}} \\ \mathsf{thrd\ p\{o.m[a\ ;\ b]\ (v)\ ;\}} \end{array} \quad \rightharpoonup \quad \begin{array}{l} \mathsf{adv\ a(x) :\ldots\ \{}\vec{C}\mathsf{\}} \\ \mathsf{thrd\ p\{}\vec{C}[^{\mathsf{p}}/_{\mathsf{this}}, {}^{\mathsf{o}}/_{\mathsf{target}}, {}^{\mathsf{o.m[\emptyset\ ;\ b]}}/_{\mathsf{proceed}}, {}^{\mathsf{v}}/_{\mathsf{x}}]\mathsf{\}} \end{array}$$

Reduction proceeds as follows: First, we choose the aspect to execute, in this case a. Next, the aspect declaration is looked up to extract the advice body, in this case $\vec{B}$. Finally, some substitutions are performed. In addition to the formal parameters, this is bound to the sender of the message p, and target is bound to the recipient o. Most significantly, $\mathsf{proceed}$ is bound to a new advised method call, referencing the remaining aspects, in this case $\mathsf{o.m[\emptyset\ ;\ b]}$. Using the substitution on $\mathsf{proceed}$, the semantics walks through the advice given in the call advice set, then the advice given in the execution advice set, using global order on aspect names to determine precedence within each of the two sets. If $\mathsf{proceed}$ does not occur in an advice body, then subsequent advice is ignored. On the other hand, if a name occurs in both the call and execution advice sets, the advice body may be executed twice. An advised method call with no advice is treated as an error; it cannot reduce.

The encoding of the class-based language into the aspect calculus provides insight into the operational semantics of the aspect calculus. The translation must account for the fact that methods in the aspect calculus do not have any method bodies. Write "$\mathsf{cbl\_}c\mathsf{\_}m$" to identify a fresh name generated from class name c and method name m. Given a method definition "$\mathsf{class}\ d <: c\ \{\ldots m(\vec{x})\ \{\vec{C}\}\ldots\}$" create the advice:

$$\begin{array}{l} \mathsf{class}\ d <: c\ \{\ldots m[\emptyset\ ;\ \mathsf{cbl\_}d\mathsf{\_}m]\ldots\} \\ \mathsf{adv\ cbl\_}d\mathsf{\_}m(\vec{x}) : \mathsf{exec}(d :: m)\ \{\vec{C}[^{\mathsf{proceed}}/_{\mathsf{super}.m}]\} \end{array}$$

For this encoding to work, it must be the case that if d is a subclass of c, then the name $\mathsf{cbl\_}d\mathsf{\_}m$ precedes $\mathsf{cbl\_}c\mathsf{\_}m$ in the advice ordering. Thus, the first aspect pulled out of the

Table 3 Aspect-Based Syntax

$D, E ::= \ \ldots$	*Declaration*	$C, B ::= \ \ldots$	*Command*
$\mathsf{adv}\ a(\vec{x}) : \phi\ \{\vec{C}\}$	Advice	$\mathsf{let}\ x = o : c.m(\vec{v})\ ;$	Dynamic Message
$M ::= m[\overline{a}\,;\overline{b}]$	*Method*	$\mathsf{let}\ x = o.m[\overline{a}\,;\overline{b}]\ (\vec{v})\ ;$	Advised Message
$L ::= c :: m$	*Label*	$\mathsf{let}\ x = \mathsf{proceed}\ (\vec{v})\ ;$	Proceed
$\phi, \psi ::=$	*Pointcut*	Replace the dynamic message syntax from Table 1	
false	False		
$\neg\phi$	Negation		
$\phi \vee \psi$	Disjunction		
$\mathsf{call}(L)$	Call		
$\mathsf{exec}(L)$	Execution		

aspect list is the closest definition of m in the class hierarchy. Finally, note that method bodies in a class-based language do not contain proceed; thus proceed can be used to encode calls to super. If no such calls exist, then subsequent advice is not executed.

3.1 Syntax

In Table 3 we extend the grammar for declarations and commands, replace the grammar for method declarations, and define a new grammar for pointcuts. The method declaration "$m[\overline{a}\,;\overline{b}]$" no longer includes a command sequence, but rather two sets of advice; the idea is that $\overline{a}$ is executed by the *caller* (call advice), $\overline{b}$ is executed by the *callee* (execution advice). The advice declaration "$\mathsf{new}\ \mathsf{adv}\ a(\vec{x}) : \phi\ \{\vec{B}\}\,;\vec{C}$" binds a, with scope $\vec{B}$ and $\vec{C}$, and also binds $\vec{x}$, this and target, with scope $\vec{B}$.

Pointcuts are used to indicate the set of methods to which advice should be attached. A point cut ϕ allows one to specify a calling point "$\mathsf{call}(c :: m)$", an execution point "$\mathsf{exec}(c :: m)$", or a combination thereof. The full set of boolean connectives can prove useful, given that point cuts apply not only to the specified class, but to all subclasses as well; negation can be used to change this.

An advised method call "$\mathsf{let}\ x = o.m[\overline{a}\,;\overline{b}]\ (\vec{v})\ ;$" indicates the collections of call advice $\overline{a}$ and execution advice $\overline{b}$ yet to be performed; the method name m is ignored. Source programs need not contain advised method calls; rather advised method calls are included because they arise during the dynamics of programs. An advised call with no advice "$\mathsf{let}\ x = o.m[\emptyset\ ;\ \emptyset]\ (\vec{v})\ ;$" is unable to reduce. The proceed command "$\mathsf{let}\ x = \mathsf{proceed}\ (\vec{v})\ ;$" causes control to advance to the next named advice, where the global order on names ($n \prec m$) is used to determine which advice is next.

3.2 Dynamic Semantics

The semantics of pointcuts is defined in Table 4. We write "$\overline{D} \vdash c :: m \in \mathrm{execadv}(\phi)$" when pointcut ϕ applies to the execution of method m in class c, and similarly for call

Table 4 Semantics of Pointcuts

$$\frac{\bar{D} \ni \mathsf{class}\ c <: d\ \{\_\}}{\bar{D} \vdash c <: d}\ (\text{S-EXTENDS}) \qquad \frac{}{\bar{D} \vdash c <: c}\ (\text{S-REFLEX}) \qquad \frac{\bar{D} \vdash c <: d \quad \bar{D} \vdash d <: e}{\bar{D} \vdash c <: e}\ (\text{S-TRANS})$$

$$\frac{\bar{D} \vdash c <: d}{D \vdash c :: m \in \text{execadv}(\mathsf{exec}(d :: m))}\ (\text{PC-EXEC}) \qquad \frac{\bar{D} \vdash c <: d}{D \vdash c :: m \in \text{calladv}(\mathsf{call}(d :: m))}\ (\text{PC-CALL})$$

$$\frac{\bar{D} \vdash L \notin \text{execadv}(\phi)}{\bar{D} \vdash L \in \text{execadv}(\neg\phi)}\ (\text{PC-ENOT}) \qquad \frac{\bar{D} \vdash L \in \text{execadv}(\phi)}{\bar{D} \vdash L \in \text{execadv}(\phi \vee \psi)}\ (\text{PC-EORL}) \qquad \frac{\bar{D} \vdash L \in \text{execadv}(\psi)}{\bar{D} \vdash L \in \text{execadv}(\phi \vee \psi)}\ (\text{PC-EORR})$$

$$\frac{\bar{D} \vdash L \notin \text{calladv}(\phi)}{\bar{D} \vdash L \in \text{calladv}(\neg\phi)}\ (\text{PC-CNOT}) \qquad \frac{\bar{D} \vdash L \in \text{calladv}(\phi)}{\bar{D} \vdash L \in \text{calladv}(\phi \vee \psi)}\ (\text{PC-CORL}) \qquad \frac{\bar{D} \vdash L \in \text{calladv}(\psi)}{\bar{D} \vdash L \in \text{calladv}(\phi \vee \psi)}\ (\text{PC-CORR})$$

pointcuts. The definition relies on a notion of subtyping "$\bar{D} \vdash c <: d$", given in the same table. Note that these definitions ignore the advice sets declared by methods.

The semantics of aspect programs is defined in Table 5. Rather than use the semantics of pointcuts directly, the rules for method invocation (R_A-DYN-MSG) and (R_A-STC-MSG), rely on the advice sets declared by methods. We do this to emulate realistic advice lookup, which should be be based on the class hierarchy alone. The more naive approach would require that each method dispatch lock all advice in the heap; our semantics is intended to be efficiently implementable. Write "$\bar{D} \vdash \text{advice}(c :: m) = [\bar{a}\,;\bar{b}]$" if $\bar{a}$ (*resp.*$\bar{b}$) is the call advice (*resp.* execution advice) declared for m in c. The definition is also given in Table 5. The rule (L_A-TOP) is required to ensure that (R_A-DYN-MSG) always succeeds in looking up execution advice, even if the method m is not defined. This is required for consistency with the woven program, where call advice is executed even if the object o does not exist. Note that (R_A-DYN-MSG) looks up the call and execution advice at different types. The rule (R_A-ADV-MSG1) describes the reduction of execution advice. The rule (R_A-ADV-MSG2) describes the reduction of call advice.

Clearly, the advice that appears in a method declaration must be consistent with that which is attached to a pointcut. We formalize this intuition as *coherence* and define a function *close* which creates coherent declaration sets. To maintain coherence, the rule for inner declarations (R_A-DEC) uses *close* to saturate the declaration set with new classes and advice.

DEFINITION 1 (COHERENCE). A collection of declarations $\bar{D}$ is *coherent* (resp. *semi-coherent*) if whenever $\bar{D} \ni \mathsf{adv}\ b(\_) : \phi\ \{\_\}$ and $\bar{D} \ni \mathsf{class}\ c <: \_\ \{\ldots m[\bar{a}\,;\,\bar{b}] \ldots\}$ then

$$b \in \bar{a} \text{ iff (resp. implies) } \bar{D} \vdash c :: m \in \text{calladv}(\phi)$$
$$\text{and } b \in \bar{b} \text{ iff (resp. implies) } \bar{D} \vdash c :: m \in \text{execadv}(\phi)$$

Table 5 Aspect-Based Reduction

Include all rules from Table 2, except (R_C-DEC), (R_C-DYN-MSG) and (R_C-STC-MSG).

$$\frac{}{\bar{D} \vdash \text{advice}(\mathsf{Object} :: m) = [\emptyset\,;\,\emptyset]}\ (L_A\text{-TOP})$$

$$\frac{\bar{D} \ni \mathsf{class}\ c <: \_\ \{\, \bar{M},\ m[\bar{a}\,;\bar{b}]\ \}}{\bar{D} \vdash \text{advice}(c :: m) = [\bar{a}\,;\bar{b}]}\ (L_A\text{-THIS})$$

$$\frac{\bar{D} \vdash \text{advice}(d :: m) = [\bar{a}\,;\bar{b}] \quad \bar{M} \not\ni m(\_)\ \{\_\} \quad \bar{D} \ni \mathsf{class}\ c <: d\ \{\bar{M}\}}{\bar{D} \vdash \text{advice}(c :: m) = [\bar{a}\,;\bar{b}]}\ (L_A\text{-SUPER})$$

$$\frac{\text{domains of } \bar{D} \text{ and } \bar{E} \text{ are disjoint}}{\begin{array}{l}(\bar{D} \vdash \bar{H},\ \mathsf{thrd}\ p\{\mathsf{new}\ \bar{E}\,;\vec{C}\}) \\ \twoheadrightarrow\ (\text{close}(\bar{D}, \bar{E}) \vdash \bar{H},\ \mathsf{thrd}\ p\{\vec{C}\})\end{array}}\ (R_A\text{-DEC})$$

$$\frac{\bar{H} \ni \mathsf{obj}\ o : d\ \{\_\} \quad \bar{D} \vdash \text{advice}(c :: m) = [\bar{a}\,;\_\,] \quad \bar{D} \vdash \text{advice}(d :: m) = [\_\,;\,\bar{b}]}{\begin{array}{l}(\bar{D} \vdash \bar{H},\ \mathsf{thrd}\ p\{\mathsf{let}\ x = o : c.m(\vec{v})\,;\vec{C}\}) \\ \twoheadrightarrow\ (\bar{D} \vdash \bar{H},\ \mathsf{thrd}\ p\{\mathsf{let}\ x = o.m[\bar{a}\,;\bar{b}]\,(\vec{v})\,;\vec{C}\})\end{array}}\ (R_A\text{-DYN-MSG})$$

$$\frac{\bar{D} \vdash \text{advice}(c :: m) = [\_\,;\,\bar{b}]}{\begin{array}{l}(\bar{D} \vdash \bar{H},\ \mathsf{thrd}\ p\{\mathsf{let}\ x = o.c :: m(\vec{v})\,;\vec{C}\}) \\ \twoheadrightarrow\ (\bar{D} \vdash \bar{H},\ \mathsf{thrd}\ p\{\mathsf{let}\ x = o.m[\emptyset\,;\,\bar{b}]\,(\vec{v})\,;\vec{C}\})\end{array}}\ (R_A\text{-STC-MSG})$$

$$\frac{\bar{D} \ni \mathsf{adv}\ b(\vec{x}) :\_\ \{\vec{B}\}}{\begin{array}{l}(\bar{D} \vdash \bar{H},\ \mathsf{thrd}\ p\{\mathsf{let}\ x = o.m[\emptyset\,;\,b,\bar{b}]\,(\vec{v})\,;\vec{C}\}) \\ \twoheadrightarrow\ (\bar{D} \vdash \bar{H},\ \mathsf{thrd}\ p\{\mathsf{let}\ x = o\{\vec{B}[{}^{o}/_{\mathsf{this}},\ {}^{o.m[\emptyset\,;\,\bar{b}]}/_{\mathsf{proceed}},\ {}^{\vec{v}}/_{\vec{x}}]\}\,;\vec{C}\})\end{array}}\ b \prec \bar{b}\ (R_A\text{-ADV-MSG1})$$

$$\frac{\bar{D} \ni \mathsf{adv}\ a(\vec{x}) :\_\ \{\vec{B}\}}{\begin{array}{l}(\bar{D} \vdash \bar{H},\ \mathsf{thrd}\ p\{\mathsf{let}\ x = o.m[a, \bar{a}\,;\bar{b}]\,(\vec{v})\,;\vec{C}\}) \\ \twoheadrightarrow\ (\bar{D} \vdash \bar{H},\ \mathsf{thrd}\ p\{\mathsf{let}\ x = p\{\vec{B}[{}^{p}/_{\mathsf{this}},\ {}^{o}/_{\mathsf{target}},\ {}^{o.m[\bar{a}\,\bar{b}]}/_{\mathsf{proceed}},\ {}^{\vec{v}}/_{\vec{x}}]\}\,;\vec{C}\})\end{array}}\ a \prec \bar{a}\ (R_A\text{-ADV-MSG2})$$

DEFINITION 2 (CLOSE). We define the function close($\overline{D}$), which saturates class declarations with advice:

$$\frac{\overline{D} \text{ is coherent}}{\text{close}(\overline{D}) = \overline{D}} \text{ (C-FIX)}$$

$$\frac{\begin{array}{l}\overline{D} \ni \mathsf{adv}\ a(\_) : \phi\ \{\ \_\ \} \\ \overline{D} \vdash c :: m \in \text{calladv}(\phi) \\ \overline{D} = \overline{E}, \mathsf{class}\ c <: d\ \{\overline{M}, m[\overline{a}; \overline{b}]\ \}\end{array}}{\text{close}(\overline{D}) = \text{close}(\overline{E}, \mathsf{class}\ c <: d\ \{\overline{M}, m[\overline{a}; \overline{b}, b]\ \})} \text{ (C-CALL)}$$

$$\frac{\begin{array}{l}\overline{D} \ni \mathsf{adv}\ b(\_) : \phi\ \{\ \_\ \} \\ \overline{D} \vdash c :: m \in \text{execadv}(\phi) \\ \overline{D} = \overline{E}, \mathsf{class}\ c <: d\ \{\overline{M}, m[\overline{a}; \overline{b}]\ \}\end{array}}{\text{close}(\overline{D}) = \text{close}(\overline{E}, \mathsf{class}\ c <: d\ \{\overline{M}, m[\overline{a}, a; b]\ \})} \text{ (C-EXEC)}$$

LEMMA 1 (CLOSE). *If $\overline{D}$ is semi-coherent, then* close($\overline{D}$) *is coherent.*

LEMMA 2 (COHERENCE PRESERVATION). *Coherence is preserved by reduction.*

Note that any program where each class declaration is taken from the class-based language is semi-coherent by construction.

4 Weaving

The weaving algorithm translates aspect-based programs into programs in the class-based language. The algorithm is not novel, being closely modeled on that used by AspectJ. Rather our contribution is that we have developed a specification of the correctness of *any* weaving algorithm.

Our goal is to show that the weaving algorithm preserves transitions made by the source aspect program. We achieve this up to a trivial renaming on methods ($\simeq$) defined below. Corectness is formalized by demanding that the following diagram can be completed.

$$\begin{array}{ccc} P & \overset{\mathit{weave}}{\Longrightarrow} & Q \\ \downarrow & & \\ P' & & \end{array} \quad \text{as} \quad \begin{array}{ccc} P & \overset{\mathit{weave}}{\Longrightarrow} & Q \\ \downarrow & & \downarrow \\ P' & \overset{\mathit{weave}}{\Longrightarrow} \simeq & Q' \end{array}$$

We also expect that a woven program not have spurious new reductions. This is formalized by demanding that the following diagram can be completed.

$$\begin{array}{ccc} P & \overset{\mathit{weave}}{\Longrightarrow} & Q \\ & & \downarrow \\ & & Q' \end{array} \quad \text{as} \quad \begin{array}{ccc} P & \overset{\mathit{weave}}{\Longrightarrow} & Q \\ \downarrow & & \downarrow \\ P' & \overset{\mathit{weave}}{\Longrightarrow} \simeq & Q' \end{array}$$

4.1 Weaving as Macro Expansion

In order to motivate the ideas, we first describe a macro-expansion approach to weaving, limiting our attention to execution pointcuts. Recall that weaving is intended as a compile-time process. Thus, the weaving process works through the entire program text. The effect of the weaving process will be to change method calls to incorporate all of the aspects advising the method.

Read $\overline{D} \vdash \text{weave}(\cdot)$ as the weaving of program fragment $(\cdot)$ in the context of declarations $\overline{D}$. The heart of the macro expansion approach to weaving is the following rule. The body of a method is determined by selecting the body of the first advice named in the advice list. The rule is applied again, after substituting the remaining advice through for proceed. Note that in the case that the advice list $\overline{b}$ is empty, then any calls to proceed will be blocked in the consequent.

$$\frac{\begin{array}{l}\overline{D} \ni \mathsf{adv}\ b(\vec{x}) : \_\{\vec{C}\} \\ \overline{D} \vdash \text{weave}() = \vec{C}'\end{array}}{\overline{D} \vdash \text{weave}(m[\emptyset\ ;\ b, \overline{b}]) = m(\vec{x})\ \{\vec{C}'\}}\ \overline{b} \neq \emptyset$$

This treatment directly captures the idea from the dynamic semantics that a call to proceed is a call to the succeeding aspect in the aspect list.

This implementation of weaving is not useful in practice because it is not guaranteed to terminate. Since weaving is intended to occur at compile time, non-termination is a bad thing.

4.2 Weaving by Introducing New Methods

We now describe a practical weaving algorithm which mimics macro expansion using run-time method invocation. Our algorithm closely follows that of AspectJ. Intuitively, given a method m affected by advice $\overline{a}$, we create an auxiliary method for each suffix of the list $\overline{a}$. Call advised methods are placed in the class of the caller, whereas execution advised methods are placed in the class of the callee.

We begin with an example in the aspect language, showing the reduction of a dynamically dispatched message. Consider the following declarations:

```
obj p : Main {}
class Main{ m[∅ ; ma] }
adv ma() : exec(Main :: m) {let x = o : c.m() ; return () ; }
obj o : c {}
class c{ m[ca ; cb] }
adv ca() : call(c :: m) {let y = proceed() ; return () ; }
adv cb() : exec(c :: m) {return () ; }
```

In the presence of these declarations, we can observe the following reductions:

```
                    thrd p{let x=o:c.m();}
(R_A-DYN-MSG) ⇀ thrd p{let x=o.m[ca ; cb]();}
(R_A-ADV-MSG2) ⇀ thrd p{let x=p{let y=o.m[∅ ; cb]();return ();};}
(R_A-ADV-MSG1) ⇀ thrd p{let x=p{let y=o{return ();};return ();};}
(R_A-RETURN) ⇀ thrd p{let x=p{let y=();return ();};}
(R_A-VAL) ⇀ thrd p{let x=p{return ();};}
(R_A-RETURN) ⇀ thrd p{let x=();}
(R_A-VAL) ⇀ thrd p{}
```

Weaving the declarations produces:

```
obj p:Main {}
class Main{m() {skip;let x=this.call_ca_m(o);return ();}
           exec_ma() {skip;let x=this.call_ca_m(o);return ();}
           call_ca_m(z) {let y=z.m();return ();}}
obj o:c {}
class c{m() {return ();}
        exec_cb() {return ();}}
```

Here "skip; $\vec{C}$" is defined as "let x=x;$\vec{C}$", where x does not appear free in $\vec{C}$. The resulting class-based reductions are as follows:

```
                    thrd p{skip;let x=p.call_ca_m(o);}
(R_C-VAL) → thrd p{let x=p.call_ca_m(o);}
(R_C-DYN-MSG) → thrd p{let x=p{let y=o.m();return ();};}
(R_C-DYN-MSG) → thrd p{let x=p{let y=o{return ();};return ();};}
(R_C-RETURN) → thrd p{let x=p{let y=();return ();};}
(R_C-VAL) → thrd p{let x=p{return ();};}
(R_C-RETURN) → thrd p{let x=();}
(R_C-VAL) → thrd p{}
```

It is worth noting several things in this example. First, the method exec_ma is introduced into Main, corresponding to the execution advice on m in Main. In addition, call_ca_m is introduced into Main and exec_cb is introduced into c, corresponding to call and execution advice on m in c. Second, note that in call_ca_m, the call to proceed has been replaced with a dynamically dispatched call on m, sent to the extra parameter z. Since woven call advice is not defined in the target object's class, the target object must be passed using this additional parameter. Finally, note the gratuitous use of "skip;"; the extra reduction is required to match the advice lookup step (R_A-DYN-MSG) in the aspect language.

The definition of weaving is split over two tables. Table 6 gives the rules for handling execution advice; Table 7 gives the rules for handling call advice. The definition proceeds by structural induction on program fragments. For clarity, we use different names when weaving different syntactic categories: *weave* for programs, *wdec* for declarations, *wheap* for heaps, *wmth* for methods, and *wstack* for stacks. Each of these is a total function on the respective domains.

Table 6 Execution Advice Weaving

$$\frac{\begin{array}{l}\text{close}(\bar{D}) \vdash \text{wdec}(\text{close}(\bar{D})) = \bar{D}' \\ \text{close}(\bar{D}) \vdash \text{wheap}(\bar{H}) = \bar{H}'\end{array}}{\text{weave}(\bar{D} \vdash \bar{H}) = (\bar{D}' \vdash \bar{H}')} \quad \text{(W-PROG)}$$

$$\frac{}{\bar{D} \vdash \text{wdec}(\mathsf{adv}\ \_(\_) : \_\ \{\_\}) = \emptyset} \quad \text{(W-ADVICE)}$$

$$\frac{}{\bar{D} \vdash \text{wheap}(\mathsf{obj}\ o : c\ \{\bar{F}\}) = \mathsf{obj}\ o : c\ \{\bar{F}\}} \quad \text{(W-OBJECT)}$$

$$\frac{\bar{D} \vdash \text{wmth}(\bar{M}) = \bar{M}'}{D \vdash \text{wdec}(\mathsf{class}\ c <: d\ \{M\}) = \mathsf{class}\ c <: d\ \{M'\}} \quad \text{(W-CLASS)}$$

$$\frac{\bar{D} \vdash \text{wstack}(p\{S\}) = (\bar{M}\ ;\ S')}{D \vdash \text{wheap}(\mathsf{thrd}\ p\{S\}) = \mathsf{thrd}\ p\{S'\}} \quad \text{(W-THREAD)}$$

$$\frac{\begin{array}{l}\bar{D} \vdash \text{genExecMth}(\bar{b}) = \bar{M} \\ \bar{M} \ni \mathsf{exec\_}\bar{b}(\vec{x})\ \{\vec{C}\}\end{array}}{\bar{D} \vdash \text{wmth}(m[\bar{a}\ ;\bar{b}]) = \bar{M}, m(\vec{x})\ \{\vec{C}\}} \quad \text{(W-METHOD)}$$

$$\frac{\begin{array}{l}\bar{D} \ni \mathsf{adv}\ b(\vec{x}) : \_\ \{\vec{C}\} \\ \bar{D} \vdash \text{wstack}(\mathsf{this}\{\vec{C}[^{\mathsf{this}.m[\emptyset\ ;\bar{b}']}/_{\mathsf{proceed}}]\}) = (\bar{M}'\ ;\ \vec{C}') \\ \bar{D} \vdash \text{genExecMth}(\bar{b}') = \bar{M}\end{array}}{\bar{D} \vdash \text{genExecMth}(b) = \bar{M}, \bar{M}', \mathsf{exec\_}\bar{b}(\vec{x})\ \{\vec{C}'\}} \quad \begin{array}{l}\bar{b} = \underline{b}, \bar{b}' \\ b \prec \bar{b}\end{array} \quad \text{(GEN-EXEC)}$$

$$\frac{\begin{array}{l}\bar{D} \vdash \text{wstack}(q\{S\}) = (\bar{M}\ ;\ S') \\ \bar{D} \vdash \text{wstack}(p\{\vec{C}\}) = (\bar{M}'\ ;\ \vec{C}')\end{array}}{\begin{array}{l}\bar{D} \vdash \text{wstack}(p\{\mathsf{let}\ x = q\{S\}\ ;\vec{C}\}) \\ \qquad = (\bar{M}, \bar{M}'\ ;\ \mathsf{let}\ x = q\{S'\}\ ;\vec{C}')\end{array}} \quad \text{(W-LET)}$$

$$\frac{\bar{D} \vdash \text{advice}(c :: m) = [\emptyset\ ;\ \_] \quad \bar{D} \vdash \text{wstack}(p\{\vec{C}\}) = (\bar{M}\ ;\ \vec{C}')}{\begin{array}{l}\bar{D} \vdash \text{wstack}(p\{\mathsf{let}\ x = o : c.m(\vec{v})\ ;\vec{C}\}) \\ \qquad = (\bar{M}\ ;\ \mathsf{skip};\mathsf{let}\ x = o.m(\vec{v})\ ;\vec{C}')\end{array}} \quad \text{(W-DYN-MSG1)}$$

$$\frac{\begin{array}{l}\text{close}(\bar{D}, \bar{E}) \vdash \text{wdec}(\text{close}(\bar{E})) = \bar{E}' \\ \text{close}(\bar{D}, \bar{E}) \vdash \text{wstack}(p\{\vec{C}\}) = (\bar{M}\ ;\ \vec{C}')\end{array}}{\bar{D} \vdash \text{wstack}(p\{\mathsf{new}\ \bar{E}\ ;\vec{C}\}) = (\bar{M}\ ;\ \mathsf{new}\ \bar{E}'\ ;\vec{C}')} \quad \text{(W-DEC)}$$

$$\frac{\bar{D} \vdash \text{wstack}(p\{\vec{C}\}) = (\bar{M}\ ;\ \vec{C}')}{\begin{array}{l}\bar{D} \vdash \text{wstack}(p\{\mathsf{let}\ x = o.c :: m(\vec{v})\ ;\vec{C}\}) \\ \qquad = (\bar{M}\ ;\ \mathsf{skip};\mathsf{let}\ x = o.c :: m(\vec{v})\ ;\vec{C}')\end{array}} \quad \text{(W-STC-MSG)}$$

$$\frac{\begin{array}{l}\bar{D} \vdash \text{wheap}(\bar{H}) = \bar{H}' \\ \bar{D} \vdash \text{wstack}(p\{\vec{C}\}) = (\bar{M}\ ;\ \vec{C}')\end{array}}{\bar{D} \vdash \text{wstack}(p\{\mathsf{new}\ \bar{H}\ ;\vec{C}\}) = (\bar{M}\ ;\ \mathsf{new}\ \bar{H}'\ ;\vec{C}')} \quad \text{(W-HEAP)}$$

$$\frac{\bar{D} \vdash \text{wstack}(p\{\vec{C}\}) = (\bar{M}\ ;\ \vec{C}')}{\begin{array}{l}\bar{D} \vdash \text{wstack}(p\{\mathsf{let}\ x = o.m[\emptyset\ ;\ \bar{b}]\ (\vec{v})\ ;\vec{C}\}) \\ \qquad = (\bar{M}\ ;\ \mathsf{let}\ x = o.\mathsf{exec\_}\bar{b}(\vec{v})\ ;\vec{C}')\end{array}} \quad \text{(W-ADV-MSG1)}$$

$$\frac{\begin{array}{l}\text{no other command rules applies} \\ \bar{D} \vdash \text{wstack}(p\{\vec{C}\}) = (\bar{M}\ ;\ \vec{C}')\end{array}}{\bar{D} \vdash \text{wstack}(p\{B\ \vec{C}\}) = (\bar{M}\ ;\ B\ \vec{C}')} \quad \text{(W-OTHER)}$$

$$\frac{}{\bar{D} \vdash \text{wstack}(p\{\}) = (\emptyset\ ;\ \emptyset)} \quad \text{(W-NONE)}$$

Table 7 Call Advice Weaving

(W-DYN-MSG2)
$$\frac{\bar{D} \vdash \text{advice}(c :: m) = [\,\bar{a}\,;\_\,] \quad \bar{D} \vdash \text{genCallMth}(m;\ \bar{a}) = \bar{M} \quad \bar{D} \vdash \text{wstack}(p\{\vec{C}\}) = (\bar{M}'\,;\ \vec{C}')}{\bar{D} \vdash \text{wstack}(p\{\,\mathsf{let}\ x = o : c.m(\vec{v})\,;\vec{C}\,\}) = (\bar{M}, \bar{M}'\,;\ \mathsf{skip}; \mathsf{let}\ x = p.\mathsf{call}\_\bar{a}\ m(o,\vec{v})\ \vec{C}')} \ \bar{a} \neq \emptyset$$

(W-ADV-MSG2)
$$\frac{\bar{D} \vdash \text{genCallMth}(m;\ \bar{a}) = \bar{M} \quad \bar{D} \vdash \text{wstack}(p\{\vec{C}\}) = (\bar{M}'\,;\ \vec{C}')}{\bar{D} \vdash \text{wstack}(p\{\,\mathsf{let}\ x = o.m[\,\bar{a}\,;\_\,]\,(\vec{v})\ \vec{C}\,\}) = (\bar{M}, \bar{M}'\,;\ \mathsf{let}\ x = p.\mathsf{call}\_\bar{a}\ m(o,\vec{v})\ \vec{C}')} \ \bar{a} \neq \emptyset$$

(GEN-CALL1)
$$\frac{\bar{D} \ni \mathsf{adv}\ a(\vec{x}) : \_\,\{\vec{C}\} \quad \bar{D} \vdash \text{wstack}(\mathsf{this}\{\vec{C}[^{y}/_{\mathsf{target}}, {}^{y.m}/_{\mathsf{proceed}}]\}) = (\bar{M}'\,;\ \vec{C}')}{\bar{D} \vdash \text{genCallMth}(m;\ a) = \bar{M}', \mathsf{call}\_a\_m(y,\vec{x})\ \{\vec{C}'\}}$$

(GEN-CALL2)
$$\frac{\bar{D} \ni \mathsf{adv}\ a(\vec{x}) : \_\,\{\vec{C}\} \quad \bar{D} \vdash \text{wstack}(\mathsf{this}\{\vec{C}[^{y}/_{\mathsf{target}}, {}^{\mathsf{this}.m[\bar{a}\,\emptyset]}/_{\mathsf{proceed}}]\}) = (\bar{M}'\,;\ \vec{C}')}{\bar{D} \vdash \text{genCallMth}(m;\ \bar{a}) = \bar{M}', \mathsf{call}\_\bar{a}\ m(y,\vec{x})\ \{\vec{C}'\}} \ \begin{array}{l}\bar{a} = a, {}^{r}a \\ a \prec \bar{a} \\ \bar{a} \neq \emptyset\end{array}$$

The resulting of weaving is a class-based program without any advice declaration; thus (W-ADVICE) returns the empty set. Instead, (W-CLASS) specifies that in a class declaration, the method bodies must be woven. This in turn causes (W-METHOD) to be applied to each method in the class. The result of weaving a method *m* is a method suite with a new method generated for each suffix of the aspect list affecting *m*. The names of the new methods are based on the declared advice; roughly speaking, the method $\mathsf{exec}\_\bar{b}$ handles the advice list $\bar{b}$.

The rule (GEN-EXEC) specifies that the body of the newly created method is given by the advice associated with the first aspect in the list, with the proceed bound to the method corresponding to the rest of the list. (GEN-EXEC) generates the methods one at a time, substituting for proceed, in each, the progressively smaller advice set. Informally, this definition can be viewed as performing the macro-expanded code described in previous subsection inside of the newly created method body. Thus, in effect, the actual expansion is postponed to runtime.

The commands in a method are woven as stacks with controlling object this; the controlling object is used only when weaving call advice. Weaving the commands in (GEN-EXEC) may produce call advised methods $\bar{M}'$. In the end, all of the collected methods are added back into the class using (W-METHOD) and (W-CLASS).

The rules for commands themselves are mostly straightforward. Note however, that (W-DYN-MSG1) and (W-STC-MSG) introduce an extra reduction, corresponding to advice lookup. Also note that (W-ADV-MSG1) substitutes $\mathsf{exec}\_\overline{b}$ for $m[\emptyset\,;\,\overline{b}]$.

The extension to call pointcuts is given in Table 7. Recall that the call aspects associated with a message are determined by the static type of the object. Apart from this difference, the weaving process for call advised methods (W-DYN-MSG2) follows the structure enunciated for execution advice. One difference stands out, however; rather than sending a message to the target *o*, call advised methods remain with the sender *p*, passing *o* as an additional parameter.

This extra parameter is substituted for target when weaving the advice body in (W-GEN-CALL1) and (W-GEN-CALL2), giving the call advice access to the target object. Note in (W-GEN-CALL2) that $\mathsf{this}.m[\overline{a}\,;\emptyset]$ is substituted through for proceed, which is later converted to $\mathsf{this}.\mathsf{call}\_\overline{a}\ m$ by (W-ADV-MSG2).

Note that if a subclass inherits a method it also inherits the associated call advice.

5 The Correctness of Weaving

Weaving is not correct for all programs. In particular weaving does not support the dynamic loading of advice that affects existing classes, although it is admissible to load classes that are affected by existing advice. Because we allow for the weaving of running threads — not something typically allowed in aspect languages — we also must make a few other sanity requirements. In particular, we require that the controlling object of all threads must be defined, and that all advised messages $m[\_\,;\,\_]$ in a thread with controlling object *p* should arise because some method defined in the class of *p* is declared to send a message to *m*. In addition, we require that programs contain no dangling references; along with the other requirements, this ensures that all of the required methods have been generated. We formalize these intuitions in the following notion of *weavability*.

DEFINITION 3 (WEAVABILITY). We define $\overline{D}; \overline{H}; \overline{n} \vdash \mathrm{weavable}(\cdot)$ on stacks in Table 8. Extend the definition to programs and advice declarations as follows:

$$\frac{\overline{D}; \overline{H}; \mathrm{bn}(\overline{H}) \vdash \mathrm{weavable}(\overline{D}) \quad \overline{D}; \overline{H}; \mathrm{bn}(\overline{H}) \vdash \mathrm{weavable}(\overline{H})}{\mathrm{weavable}(\overline{D} \vdash \overline{H})} \qquad \frac{\overline{D}; \overline{H}; \overline{n}, \vec{x}, \mathsf{this}, \mathsf{target} \vdash \mathrm{weavable}(\mathsf{this}\{\vec{C}\})}{\overline{D}; \overline{H}; \overline{n} \vdash \mathrm{weavable}(\mathsf{adv}\ a(\vec{x}) : \{\vec{C}\})}$$

Extend the definition to all other program constructs homorphically using conjunction.

LEMMA 3. *Weavability is preserved by reduction.*

Even given weavability, our definition of weaving is not quite exact with respect to the reduction semantics. As seen in the example in the last section, in the aspect language a dynamic message is converted to an advised message in one reduction. The names generated by weaving these are different in the case that there is no call advice. The discrepancy cannot be handled during weaving, since the list of execution advice cannot be determined statically. We therefore must work up to a relation that equates *m* with $\mathsf{exec}\_\overline{b}$ in the appropriate circumstances.

Table 8 Weavability of Stacks

To simplify the definition we write "$\mathrm{wheap}_{\bar{D}}(H)$" for "$\bar{D} \vdash \mathrm{wheap}(H)$", and "$\mathrm{wdec}_{\bar{D}}(E)$" for "$\bar{D} \vdash \mathrm{wdec}(E)$". Also, "write $\mathrm{bn}(\bar{H})$" for the set of object names bound by heap $\bar{H}$.

(WC-DEC)
$$\frac{\begin{array}{l}\mathrm{wheap}_{\bar{D}}(\bar{H}) = \mathrm{wheap}_{\mathrm{close}(\bar{D},\bar{D}')}(\bar{H}) \\ \mathrm{close}(\mathrm{wdec}_{\bar{D}}(\bar{D}), \mathrm{wdec}_{\mathrm{close}(\bar{D},\bar{D}')}(\mathrm{close}(\bar{D}'))) \\ \quad = \mathrm{wdec}_{\mathrm{close}(\bar{D},\bar{D}')}(\mathrm{close}(\bar{D}, \bar{D}')) \\ \mathrm{close}(\bar{D}, \bar{D}')\,;\, \bar{H};\, \bar{n} \vdash \mathrm{weavable}(p\{\vec{C}\})\end{array}}{\bar{D};\, \bar{H};\, \bar{n} \vdash \mathrm{weavable}(p\{\mathsf{new}\ \bar{D};\vec{C}\})}$$

(WC-HEAP)
$$\frac{\bar{D};\, \bar{H}, \bar{H}';\, \bar{n}, \mathrm{bn}(\bar{H}') \vdash \mathrm{weavable}(p\{\vec{C}\})}{\bar{D};\, \bar{H};\, \bar{n} \vdash \mathrm{weavable}(p\{\mathsf{new}\ \bar{H}';\vec{C}\})}$$

(WC-ADV-MSG)
$$\frac{\begin{array}{l}\bar{H} \ni \mathsf{obj}\ o : c\ \{\_\} \\ \bar{H} \ni \mathsf{obj}\ p : d\ \{\_\} \\ \mathrm{wdec}_{\mathrm{close}(\bar{D})}(\mathrm{close}(\bar{D})) \vdash \mathrm{body}(c :: \mathsf{exec}\_\bar{b})\ \text{defined} \\ \mathrm{wdec}_{\mathrm{close}(\bar{D})}(\mathrm{close}(\bar{D})) \vdash \mathrm{body}(d :: \mathsf{call}\_\bar{a}\ m)\ \text{defined} \\ \bar{D};\, \bar{H};\, \bar{n}, x \vdash \mathrm{weavable}(p\{\vec{C}\})\end{array}}{\bar{D};\, \bar{H};\, \bar{n} \vdash \mathrm{weavable}(p\{\mathsf{let}\ x = o.m[\bar{a}\bar{b}]\ (\vec{v})\ ;\vec{C}\})}$$

(WC-LET)
$$\frac{\bar{D};\, \bar{H};\, \bar{n} \vdash \mathrm{weavable}(o\{S\}) \qquad \bar{D};\, \bar{H};\, \bar{n}, x \vdash \mathrm{weavable}(p\{\vec{C}\})}{\bar{D};\, \bar{H};\, \bar{n} \vdash \mathrm{weavable}(p\{\mathsf{let}\ x = o\{S\}\vec{C}\})}$$

(WC-DYN-MSG)
$$\frac{o \in \bar{n} \qquad \bar{D};\, \bar{H};\, \bar{n}, x \vdash \mathrm{weavable}(p\{\vec{C}\})}{\bar{D};\, \bar{H};\, \bar{n} \vdash \mathrm{weavable}(p\{\mathsf{let}\ x = o : c.m(\vec{v})\ \vec{C}\})}$$

(WC-OTHER1)
$$\frac{\text{no other let rule applies} \qquad \bar{D};\, \bar{H};\, \bar{n}, x \vdash \mathrm{weavable}(p\{\vec{C}\})}{\bar{D};\, \bar{H};\, \bar{n} \vdash \mathrm{weavable}(p\{\mathsf{let}\ x = \ldots\vec{C}\})}$$

(WC-STC-MSG)
$$\frac{o \in \bar{n} \qquad \bar{D};\, \bar{H};\, \bar{n}, x \vdash \mathrm{weavable}(p\{\vec{C}\})}{\bar{D};\, \bar{H};\, \bar{n} \vdash \mathrm{weavable}(p\{\mathsf{let}\ x = o.c :: m(\vec{v})\ \vec{C}\})}$$

(WC-OTHER2)
$$\frac{\text{no other command rule applies} \qquad \bar{D};\, \bar{H};\, \bar{n}, x \vdash \mathrm{weavable}(p\{\vec{C}\})}{\bar{D};\, \bar{H};\, \bar{n} \vdash \mathrm{weavable}(p\{B\vec{C}\})}$$

DEFINITION 4 (NAME EQUIVALENCE). Let $\simeq$ be the equivalence on class-based commands generated by:

$$\frac{\bar{H} \ni \mathsf{obj}\ o : d\ \{\_\} \qquad \bar{D} \vdash \mathrm{advice}(d :: m) = [\_;\, \bar{b}]}{D;\, H \vdash \mathsf{let}\ x = o.m(\vec{v})\ ; \simeq \mathsf{let}\ x = o.\mathsf{exec}\_b(\vec{v})\ ;}$$

$$\frac{\bar{D} \vdash \mathrm{advice}(c :: m) = [\_;\, \bar{b}]}{\bar{D};\, \bar{H} \vdash \mathsf{let}\ x = o.c :: m(\vec{v})\ ; \simeq \mathsf{let}\ x = o.\mathsf{exec}\_b(\vec{v})\ ;}$$

$$\frac{C \text{ is not a method call}}{\bar{D};\, \bar{H} \vdash C \simeq C}$$

Extend the definition to all other program constructs homorphically using conjunction. Let $P = (\bar{D} \vdash \bar{H})$ and $P' = (\bar{D}' \vdash \bar{H}')$. We write "$P \simeq P'$" when $\bar{D};\, \bar{H} \vdash P \simeq P'$ and $\bar{D}';\, \bar{H}' \vdash P \simeq P'$.

THEOREM 1. *Suppose that an aspect-based program P is coherent and weavable, and that $P \twoheadrightarrow P'$. Then there exists some Q', such that* $\mathrm{weave}(P) \twoheadrightarrow Q'$ *and* $Q' \simeq \mathrm{weave}(P')$.

Suppose that an aspect-based program P is coherent and weavable, and that $\text{weave}(P) \rightarrow Q'$. *Then there exists some* P', *such that* $P \twoheadrightarrow P'$ *and* $P' \simeq \text{weave}(Q')$.

6 Related work

We refer the reader to the October 2001 issue of CACM for a comprehensive survey and references to the range of approaches and applications of AOP. Here, we restrict ourselves to the several recent efforts to formalize and provide simple conceptual models of some features of aspect-oriented languages.

There are several efforts focused largely on weaving and the understanding of pointcuts. For example, The Aspect SandBox [10] provides a testbed to experiment with weaving strategies. Wand, Kiczales, and Dutchyn [21], give a denotational semantics for a mini-language that embodies the key features of dynamic join points, pointcut designators, and advice. R. Douence and O. Motelet and M. Südholt [8] describe a domain-specific language for the definition of crosscuts and sketch a prototype implementation in Java which has been systematically derived from the language definition. H. Masuhara, Kiczales and Dutchyn [17] present a semantics-based compilation framework for an aspect-oriented programming language. Using partial evaluation, the framework studies which aspects can be woven in at compile time and which dispatches must be executed at run-time.

In contrast to this line of research, our aim has been to develop an independent *specification* of weaving. We have taken the point of view that the operational semantics of the aspect language validates a given implementation of weaving. In this sense, our approach is complementary to this body of work. One might daresay that a suitable mixture of these ideas could result in a model of a real-life aspect-oriented programming language.

The research closest to the spirit of our paper is the concurrent and independent work of Walker, Zdancewic and Ligatti [20]. This paper studies a powerful core calculus of aspects, not including subtyping, where both advice and join-points are first class entities that can be created and manipulated at runtime. On the one hand, their paper proves a type soundness theorem for a calculus with features that are not available in our core calculus. On the other hand, their study focuses on the case of execution pointcuts by assuming that the source code of the advised method is available for transformation.

From a more foundational viewpoint, Meuter [18] describe a view of aspects as monads. In this view, the weaver then becomes a lifter to transform programs through different monads. Andrews [3] views aspects in a process-algebraic context. Both these papers can be viewed as attempts to translate aspects into other frameworks. In contrast, our work follows the line of research into object calculi and their adaptations to particular programming languages such as Java. In this spirit, we study aspects as a primitive computational entity in their own right.

References

1. Martin Abadi and Luca Cardelli. *A theory of objects*. Springer-Verlag, 1996.
2. M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object-interactions using composition-filters. In *In object-based distributed processing, LNCS*, 1993.
3. J. Andrews. Process-algebraic foundations of aspectoriented programming. In *In Reflection, LNCS 2192*, 2001.
4. L. Bergmans. *"Composing Concurrent Objects - Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs"*. Ph.d. thesis, University of Twente, 1994. http://wwwhome.cs.utwente.nl/ bergmans/phd.htm.
5. Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155, 1999. an extended abstract appeared in Proceedings of TACS '97, LNCS 1281, Springer-Verlag, pp. 415-438.
6. Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *European Conference on Object-Oriented Programming (ECOOP)*, 1997.
7. Kim B. Bruce, Adrian Fiech, Angela Schuett, and Robert van Gent. A type-safe polymorphic object-oriented language. In *European Conference on Object-Oriented Programming (ECOOP)*, 1995.
8. R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of the 3rd International Conference on Reflection and Crosscutting Concerns*, LNCS. Springer Verlag, September 2001. long version is http://www.emn.fr/info/recherche/publications/RR01/01-3-INFO.ps.gz.
9. Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the java type system sound? *Theory and Practice of Object Systems*, 5(11):3–24, 1999.
10. Christopher Dutchyn, Gregor Kiczales, and Hidehiko Masuhara. `http://www.cs.ubc.ca/labs/spl/projects/asb.html`.
11. Kathleen Fisher, John Reppy, and Jon G. Riecke. A calculus for compiling and linking classes. In *European Conference on Object-Oriented Programming (ECOOP)*, 2000.
12. Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of OOPSLA*, October 1999. Full version in ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), May 2001.
13. Atsushi Igarashi and Benjamin C. Pierce. On inner classes. In *European Conference on Object-Oriented Programming (ECOOP)*, 2000. Also in informal proceedings of the Seventh International Workshop on Foundations of Object-Oriented Languages (FOOL). To appear in *Information and Computation*.
14. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
15. 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 (ECOOP)*, 1997.
16. K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.
17. Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs.
18. W. De Meuter. Monads as a theoretical foundation for aop. In *International Workshop on Aspect-Oriented Programming at ECOOP*, 1997.

19. H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2001.
20. David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. Submitted for publication.
21. Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. appeared in Informal Workshop Record of FOOL 9, pages 67-88; also presented at FOAL (Workshop on Foundations of Aspect-Oriented Languages), a satellite event of AOSD 2002, 2002.