A Typed Calculus of 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. In earlier work, we described an untyped calculus of aspect programs with a direct description of the dynamic semantics. This calculus provides a specification for the correctness of weaving.
In this paper, we turn our attention to the interaction of aspects and types, whose subtleties are amply illustrated by the difficulties encountered by current compilers of aspect languages. We develop a typed calculus of aspect programs that includes inner classes, concurrency and dynamic arrival of new advice. To our knowledge, this is the first source-level typing system for a class-based aspect-oriented language.
We prove that types are preserved by reduction in the aspect calculus and that well-typed programs make progress. We also show that weaving preserves typability of programs by mapping well-typed aspect programs to well-typed class based programs.

1 Introduction

In this paper, we give the static semantics for an aspect-based language and prove its correctness with respect to reduction and weaving.

Aspects: A Short Introduction We begin with a short example to introduce the basic vocabulary of aspect-oriented programming and illustrate the underlying issues [3, 16, 21, 20, 17, 2]. 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());
   }}
```

* Research supported in part by National Science Foundation grants 0244901 and 0208549.

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 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*.

Typing issues in aspect languages. Quite early on, the study of expressive type systems for aspect languages was recognized as an important research problem. For example, see the report [18] on the first "Foundations of Aspect-Oriented Languages" (FOAL) workshop [19] in the types forum. In relationship to object-oriented languages, it is helpful to view advice, at a first approximation, as acting like method update [1]; hence it needs to be treated carefully. For example, the following program compiles successfully with the latest version of the AspectJ compiler (ajc1.1). However, this program yields a runtime `ClassCastException` as expected.

```
public class Foo {
  public static void main(String[] args) {
    System.out.println(new D().m());
  }}
class D {
  public String m() {
    return "D";
  }}
aspect B {
  Object around(): call(*  D.m()) {
    return new Integer(1);
  }}
```

One can speculate on the the motivation of the designers of the type system of AspectJ — presumably, the aim was to increase the reuse of advice code by permitting around advice to modify the return types to any "consistent" return type. This design aim has exposed the type system to the subtlety of the interaction of classes, polymorphism and aspects.

This paper has two motivations.

- Methodologically speaking, we aim to import the techniques and experience developed in the well-developed study of type systems for class-based language to source level typing of aspect languages. Such a treatment would provide the guarantees, such as type-safety and compilation to an interface, that one has come to expect of modern programming languages.
- In terms of language design, the study of this paper is an essential first step towards language design in the space of aspect languages — for example, we point out the AspectJ and Generic Java designs are not currently compatible, and the study of a type-safe generic extension to an aspect language requires the study carried out in this paper as a foundation.

Results in this paper. We study the static semantics of an aspect language, proving three fundamental properties:

- well-typed programs can always make progress,
- well-typing is preserved by reduction, and
- well-typing is preserved by translation to a class-based language via weaving.

Our language of pointcuts can express the call and execution of any method; we have not considered temporal operators, such as `cflow` in AspectJ. Our aspect calculus permits dynamic arrival of new advice and includes inner-classes and concurrency, providing evidence for the scalability of the results to a reasonably sized language.

In terms of the expressiveness of the typing system, we have chosen deliberately to study a simple first-order typing system. We re-emphasize our belief that polymorphism is essential for achieving satisfactory expressiveness while maintaining type safety in an aspect-oriented setting and we discuss supporting evidence in section 6. However, the more restrictive study of this paper is essential groundwork for the development of these more expressive type systems.

The Rest of this Paper The rest of this paper is organized as follows. In section 2, we provide an overview of our approach to the dynamic and static semantics of aspects, as well as weaving. In section 3, we present the details of a typed class based language. In the following section 4, we turn our attention to the typed aspect calculus. Section 5 describes the typed weaving algorithm. Finally, we discuss the interactions of aspects and polymorphism in section 6 and summarize related work in section 7. All proofs and some definitions are relegated to the appendices for the indefatigable reader.

2 Overview of Our Approach

We describe the dynamic semantics, the typing system, and weaving.

2.1 An Untyped Calculus

In our earlier work [15], we have described an untyped calculus of aspect oriented programs. In this calculus, classes are merely sets of method *names* and do not contain any code. All code is associated with named advice of the form: $\mathsf{advice}\ a(\vec{x})\ \mathsf{at}\ \phi\ \{\vec{C}\}$. The name of the advice is a; the parameters and code associated with the advice are given respectively by $\vec{x}$ and $\vec{C}$; the advice is attached to the pointcut given by ϕ. A pointcut is a set of basic call and execution pointcuts and is described syntactically as an element of a boolean algebra over these atoms.

We discuss the key cases of reduction to highlight the novel features of aspect oriented programs. A program consists of a collection $\overline{D}$ of class and advice declarations, together with a collection $\overline{H}$ of threads and objects on the heap. A method call $o{:}c.m$ on an object o of static type c expands out to an advised method call $o{:}c.m[\overline{a}\,;\overline{b}]$ to include the list of advice associated with method m of o. This advice list is looked up from the declarations $\overline{D}$. As discussed earlier, the call advice list $\overline{a}$ is based on the static type c of o; this type is deduced from the syntax of the program. The execution advice

list $\overline{b}$ is based on the true dynamic type d of o which is fetched from the heap, $\overline{H}$, of objects. Call advice is listed before execution advice. The order of advice within the call advice list (and the execution advice list) is based on a given global order on advice names.

$$\frac{\begin{array}{l}\overline{H} \ni \mathsf{object}\, o : d\, \{\_\} \\ \overline{D} \Vdash \mathsf{advice}(c :: m) = [\overline{a}\,;\_] \\ \overline{D} \Vdash \mathsf{advice}(d :: m) = [\_\,;\overline{b}]\end{array}}{\begin{array}{r}\overline{D} \vdash \overline{H}, \mathsf{thread}\, p\, \{\mathsf{let}\, X = o : c.m(\vec{v})\,;\vec{C}\,\} \\ \rightarrowtriangle\ \overline{D} \vdash \overline{H}, \mathsf{thread}\, p\, \{\mathsf{let}\, X = o : c.m[\overline{a}\,;\overline{b}]\,(\vec{v})\,;\vec{C}\,\}\end{array}}$$

The notation $p\,\{...\}$ is used to indicate that the controlling object of the context is p. The reason for this modeling will become clearer as we turn our attention to the execution of advised method calls.

We sketch the execution of advised method calls next, beginning with call advice. To execute $o : c.m[a, \overline{a}\,;\overline{b}]$, we first look up the body of a in the declarations, then substitute the "rest of the advice", namely $o : c.m[\overline{a}\,;\overline{b}]$, for $\mathsf{proceed}$ in the body $\vec{B}$. In addition, there are two other substitutions: this becomes the name of the controlling object p, and target becomes the name of the callee o. The unrolled body continues to execute under the control of p.

$$\frac{\begin{array}{l}\overline{D} \ni \mathsf{advice}\, a(\vec{x})\ \mathsf{at}\ \_\ \{\vec{B}\} \\ \vec{B}' = \vec{B}[^{o : c.m[\overline{a}\,;\overline{b}]}/_{\mathsf{proceed}}]\end{array}}{\begin{array}{r}\overline{D} \vdash \overline{H}, \mathsf{thread}\, p\, \{\mathsf{let}\, x : r = o : c.m[a, \overline{a}\,;\overline{b}]\,(\vec{v})\,;\vec{C}\,\} \\ \rightarrowtriangle\ \overline{D} \vdash \overline{H}, \mathsf{thread}\, p\, \{\mathsf{let}\, x : r = p\{\vec{B}'[^{p}/_{\mathsf{this}}, ^{o}/_{\mathsf{target}}, ^{\vec{v}}/_{\vec{x}}]\}\,;\vec{C}\,\}\end{array}}$$

Execution advice is similar in spirit to call advice. The structural differences between call and execution are modeled by the different substitutions: both this and target are substituted by the callee o, and the unrolled body executes under the control of o.

$$\frac{\begin{array}{l}\overline{D} \ni \mathsf{advice}\, b(\vec{x})\ \mathsf{at}\ \_\ \{\vec{B}\} \\ \vec{B}' = \vec{B}[^{o : c.m[\emptyset\,;\overline{b}]}/_{\mathsf{proceed}}]\end{array}}{\begin{array}{r}\overline{D} \vdash \overline{H}, \mathsf{thread}\, p\, \{\mathsf{let}\, x : r = o : c.m[\emptyset\,; b, \overline{b}]\,(\vec{v})\,;\vec{C}\,\} \\ \rightarrowtriangle\ \overline{D} \vdash \overline{H}, \mathsf{thread}\, p\, \{\mathsf{let}\, x : r = o\{\vec{B}'[^{o}/_{\mathsf{this}}, ^{o}/_{\mathsf{target}}, ^{\vec{v}}/_{\vec{x}}]\}\,;\vec{C}\,\}\end{array}}$$

There are several advantages to formalizing the aspect calculus directly. First, the source-level semantics for aspects provides a *specification* for the weaving algorithm, and enables us to prove the *correctness* of the weaving transformation. Second, this aspect calculus naturally allows for the dynamic addition of advice to a running program — in this respect, the calculus is richer than statically woven languages such as AspectJ, and in the spirit of more recent experimental efforts. Third, the calculus permits us to reduce the number of features that need to be studied by providing a formal basis to establish redundancies, e.g. method bodies are unnecessary, as are various other kinds of advice.

2.2 Onto Typing

Our calculus in [15] was untyped, e.g. field assignment was not typechecked. For the usual, and by now well-understood reasons, this permits the expression of several undesirable programs. Furthermore, the absence of typing raises issues specific to the aspect setting. Consider the reduction rule for execution advice described earlier. Suppose that $\bar{b}$ is empty and the body of the code associated with b is a call to proceed. In this case, the proceed variable is bound to an empty advice list, leading to the aspect-oriented analogue of the "unknown method" error. (AspectJ avoids this error, albeit at the cost of redundancy, since the method bodies of the underlying OO paradigm do not contain calls to proceed.)

In the light of the extensive literature on type systems for OO programming (e.g. see [1, 5, 22] for a survey of research into types for OO languages), a natural impulse could be to try to impose such a typing paradigm on the untyped aspect calculus. What should be the type associated with a piece of advice:

$$\mathsf{advice}\ a(\vec{x})\ \mathsf{at}\ \phi\ \{\vec{C}\}$$

Clearly, it has to contain type information associated with the arguments u and the return value r — this is standard — to yield:

$$\mathsf{advice}\ a(\vec{x}:\vec{u}):r\ \mathsf{at}\ \phi\ \{\vec{C}\}$$

However, the reduction rule for call advice described earlier motivates the need for more type information. In this reduction rule, the body of advice undergoes a substitution of this for the controlling object of the context p and target for the callee p. Thus, accurate typing of the advice demands type assumptions about the caller and the callee objects. This greater symmetry between caller and callee, relative to usual OO programming, reflects the incremental construction of control structures in aspect-oriented programming.

The simultaneous type assertions on callers and callees permeate our system. For example, the form of the typing judgments for pointcuts and method names are:

$$\bar{D} \Vdash \phi\ :\ \mathsf{Ptcut}(\vec{u}):r\ \mathsf{for}\ s\ \mathsf{in}\ c$$
$$\bar{D} \Vdash m\ :\ \mathsf{Mth}(\vec{u}):r\ \mathsf{for}\ s\ \mathsf{in}\ c$$

These judgment constrain the return type to be r, the type of the caller to be (a subtype of) s and the type of the callee to be (a subtype of) c.

A piece of advice is well-typed if its body is valid given correct bindings to this and target. The types of these identifiers are inherited from the type of the pointcut.

$$\frac{\bar{D} \Vdash \phi\ :\ \mathsf{Ptcut}(\vec{u}):r\ \mathsf{for}\ s\ \mathsf{in}\ t \qquad \bar{D};\ \bar{Z}, \vec{x}:\vec{u}, \mathsf{this}:s, \mathsf{target}:t \Vdash \vec{C}\ :\ \mathsf{Stk}(\vec{u}):r\ \mathsf{for}\ s}{\bar{D};\ \bar{Z} \Vdash \mathsf{advice}\ a(\vec{x}:\vec{u}):r\ \mathsf{at}\ \phi\ \{\vec{C}\}\ :\ \mathsf{Dec}}$$

We enforce consistent types on advice sets using pointcuts, which, in turn, must be consistent with the methods they name.

For a call pointcut, we require the type of the method to match up with the type of the pointcut, i.e. the return type of the method matches the return type of the pointcut, and the types of the caller and callee are subtypes of the corresponding types in the method.

$$\frac{\bar{D} \Vdash m \ : \ \mathsf{Mth}(\vec{u}) : r \ \mathsf{for}\ s' \ \mathsf{in}\ c \quad \bar{D} \vdash s' <: s \quad \bar{D} \vdash c <: t}{\bar{D} \Vdash \mathsf{call}(c :: m) \ : \ \mathsf{Ptcut}(\vec{u}) : r \ \mathsf{for}\ s \ \mathsf{in}\ t}$$

In contrast, an execution pointcut is executed under the control of the callee object. So, in this case, the caller type information in the method is ignored.

$$\frac{\bar{D} \Vdash m \ : \ \mathsf{Mth}(\vec{u}) : r \ \mathsf{for}\ \_ \ \mathsf{in}\ c \quad \bar{D} \vdash c <: s \quad \bar{D} \vdash c <: t}{\bar{D} \Vdash \mathsf{exec}(c :: m) \ : \ \mathsf{Ptcut}(\vec{u}) : r \ \mathsf{for}\ s \ \mathsf{in}\ t}$$

The idea of type restrictions on the caller of methods is directly reflected in the source language by declarations of the form:

$$\mathsf{protected}\ s\ \mathsf{method}\ m(\vec{x} : \vec{t}) : r\,;$$

Such a method can only be called by callers at subtypes of s; hence, the keyword protected, to indicate protection from callers that do not satisfy this type restriction. We permit such type annotations on fields too. This is a generalization of Java's annotations `public` (protected Object in our notation) and `protected` (protected c if the containing class is c).

Such declarations form the basis of deducing type restrictions on callers. For example, if method m is declared in class d as above, we can deduce

$$\bar{D} \Vdash m \ : \ \mathsf{Mth}(\vec{u}) : r \ \mathsf{for}\ s \ \mathsf{in}\ c$$

for any subclass c of d.

$$\frac{\bar{D} \ni \mathsf{class}\ d <: \_ \{ \_\ \bar{M} \} \quad \bar{M} \ni \mathsf{protected}\ s\ \mathsf{method}\ m(\_ : \vec{u}) : r\,; \quad \bar{D} \vdash c <: d}{\bar{D} \vdash m \ : \ \mathsf{Mth}(\vec{u}) : r \ \mathsf{for}\ s \ \mathsf{in}\ c}$$

In the technical body of this paper, we describe precisely how such judgments are propagated through the various constructs of the calculus, such as commands.

It is worth pointing out that our approach to typing the caller (bound to this in advice) is quite different than that of AspectJ. In AspectJ, the type of the this is significant in the dynamic semantics of call advice: the advice applies only if the caller inhabits the correct type. In our language, both call and execution advice are uniform across all callers. This allows a cleaner separation of the dynamics and the statics.

A type system set up using the ideas described above provides some of the guarantees that we are seeking: for example, to ensure that field updates are type-correct. However, by itself, such a type system doesn't handle the problem of ensuring that the proceed variable is always bound. Our second and final set of type annotations address this problem by distinguishing advice that does not contain a call to proceed from advice that does. We require that advice be annotated with a *placement* ρ, where ρ can be either around or replace. The final form of advice declarations is then:

$$\rho \text{ advice } a(\vec{x}:\vec{u}):r \text{ at } \phi \{\vec{C}\}$$

Advice with the replace annotation cannot contain calls to proceed whereas around annotation imposes no such restriction. For example, in AspectJ, method bodies of the underlying OO paradigm do not contain calls to proceed and can be given the replace annotation. These annotations permit us to identify valid advice lists that are guaranteed to terminate without dangling calls to proceed. This is achieved by requiring that each execution advice list contain at least one piece of replace advice.

The type system with both of these basic ingredients satisfies standard desirable properties. We prove that types are preserved by reduction in the aspect calculus and that well-typed programs make progress:

- If $\Vdash P : \mathsf{Prog}$ and $P \twoheadrightarrow^{*} P'$ then $\Vdash P' : \mathsf{Prog}$.
- If $\Vdash (\overline{D} \vdash H, \text{thread } o\,\{S\}) : \mathsf{Prog}$ then either thread $o\,\{S\}$ has terminated or $(\overline{D} \vdash \overline{H}, \text{thread } o\,\{S\}) \rightarrow (\overline{D'} \vdash \overline{H'}, \text{thread } o\,\{S'\})$.

2.3 Weaving

The aspect calculus naturally allows for the dynamic addition of advice to a running program. Clearly, programs that dynamically load advice affecting existing classes cannot be woven statically. In [15] we defined a notion of *weavability*, which excludes such programs, and we showed that for weavable programs, the weaving algorithm is correct at run-time, that is (up to some renaming of methods) we can complete:

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

and (again, up to some renaming of methods) we can complete:

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

In this paper, we show also that weaving is correct at compile-time, i.e. any well-typed aspect-based program is woven to a well-typed class-based program.

$$\text{If } \Vdash P : \mathsf{Prog} \text{ then } \vdash \text{weave}(P) : \mathsf{Prog}$$

Table 1 Class-Based Syntax

$a,..,z$	*Name*	S,T ::=	*Call Stack*
X,Y,Z ::= $n:t$	*Typed Name*	$\vec{C}$	Current Frame
P,Q ::= $(\bar{D} \vdash \bar{H})$	*Program*	let $X=o\{S\};\vec{C}$	Pushed Frame
D,E ::= class c <: d $\{\bar{F}\ \bar{M}\}$	*Declaration*	C,B ::=	*Command*
M ::= protected s method $m(\vec{X}):r\ \{\vec{C}\}$	*Method*	return v;	Return
F ::= protected s field $f:t$;	*Field Type*	let $X=v$;	Value
V ::= $f=v$;	*Field Value*	let $X=o.m(\vec{v})$;	Dynamic Message
H,G ::=	*Heap Element*	let $X=o.c::m(\vec{v})$;	Static Message
object $o:c\ \{\bar{V}\}$	Object	let $X=o.f$;	Get Field
thread $o\ \{S\}$	Thread	set $o.f=v$;	Set Field
		new $\bar{D}\ \bar{H}$;	New

3 A Class-based Language

Our class-based languages is similar in spirit to Classic Java [12], Featherweight Java [13] and Java$_s$ [9], although our language includes mutability and concurrency. This basic model has also been extended to address genericity [4, 13] and the removal of inner classes [14]. Other work has concentrated on translations of class-based languages into polymorphic λ-calculi or object-based languages [6, 1, 7, 8, 11].

NOTATION. For any metavariable x, we write $\vec{x}$ for ordered sequences, and $\bar{x}$ for unordered sequences, of x's. 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 *a*–*b* for advice names; *c*–*d* and *r*–*u* for class names; *f* for field names; *m* for method names; *o*–*q* for object reference names; *x*–*z* for variables (parameters and let-bound names); *v* for values (object references and variables).

3.1 Syntax

The syntax is in Table 1. The main novelty is in method declarations of the form

$$\text{protected } s \text{ method } m(\vec{X}):r\ \{\vec{C}\}$$

and field declarations of the form protected s field $f:t$;. The type system will ensure that these methods and fields are only accessed by subclasses of s.

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. We identify programs up to renaming of bound names and define substitution $\vec{C}[y/x]$ as usual. We define the notion

Table 2 Class-Based Reduction

$$\boxed{\bar{D} \vdash \mathrm{mbody}(c::m) = (\vec{x})\vec{C}}$$

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

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

$$\boxed{P \twoheadrightarrow^* P'}$$

$$\frac{\bar{D} \vdash \bar{H}, \mathsf{thread}\ p\ \{\ S\ \} = P \quad \twoheadrightarrow\ \bar{D}' \vdash \bar{H}', \mathsf{thread}\ p\ \{\ S'\ \} = P'}{P \twoheadrightarrow^* P'}\ (\mathrm{R_C\text{-}STEP})$$

$$\frac{}{P \twoheadrightarrow^* P}\ (\mathrm{R_C\text{-}ID}) \qquad \frac{P \twoheadrightarrow^* P' \qquad P' \twoheadrightarrow^* P''}{P \twoheadrightarrow^* P''}\ (\mathrm{R_C\text{-}TRANS})$$

$$\boxed{\bar{D} \vdash \bar{H}, \mathsf{thread}\ p\ \{S\} \twoheadrightarrow \bar{D}' \vdash \bar{H}', \mathsf{thread}\ p\ \{S'\}}$$

$$\frac{\bar{D} \vdash \bar{H}, \mathsf{thread}\ q\ \{\ S\ \} \quad \twoheadrightarrow\ \bar{D}' \vdash \bar{H}', \mathsf{thread}\ q\ \{\ S'\ \}}{\bar{D} \vdash \bar{H}, \mathsf{thread}\ p\ \{\ \mathsf{let}\ X = q\{\ S\ \}; \vec{C}\ \} \twoheadrightarrow \bar{D}' \vdash \bar{H}', \mathsf{thread}\ p\ \{\ \mathsf{let}\ X = q\{\ S'\ \}; \vec{C}\ \}}\ (\mathrm{R_C\text{-}LET})$$

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

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

$$\frac{\mathrm{domains}\ \bar{D}, \bar{E}, \bar{H}, \bar{G}\ \mathrm{disjoint}}{\bar{D} \vdash \bar{H}, \mathsf{thread}\ p\ \{\ \mathsf{new}\ \bar{E}\ \bar{G}; \vec{C}\ \} \twoheadrightarrow \bar{D}, \bar{E} \vdash \bar{H}, \bar{G}, \mathsf{thread}\ p\ \{\ \vec{C}\ \}}\ (\mathrm{R_C\text{-}NEW})$$

$$\frac{\bar{H} \ni \mathsf{object}\ o:c\ \{\ \_\ \} \qquad \bar{D} \vdash \mathrm{mbody}(c::m) = (\vec{x})\vec{B}}{\bar{D} \vdash \bar{H}, \mathsf{thread}\ p\ \{\ \mathsf{let}\ X = o.m(\vec{v})\ ; \vec{C}\ \} \twoheadrightarrow \bar{D} \vdash \bar{H}, \mathsf{thread}\ p\ \{\ \mathsf{let}\ X = o\{\ \vec{B}[o/\mathsf{this}, \vec{v}/\vec{x}]\ \}; \vec{C}\ \}}\ (\mathrm{R_C\text{-}DYN\text{-}MSG})$$

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

$$\frac{}{\bar{D} \vdash \bar{H}, \mathsf{object}\ o:c\ \{f = v; \bar{V}\}, \mathsf{thread}\ p\ \{\ \mathsf{let}\ X = o.f; \vec{C}\ \} \twoheadrightarrow \bar{D} \vdash \bar{H}, \mathsf{object}\ o:c\ \{f = v; \bar{V}\}, \mathsf{thread}\ p\ \{\ \mathsf{let}\ X = v; \vec{C}\ \}}\ (\mathrm{R_C\text{-}GET})$$

$$\frac{}{\bar{D} \vdash \bar{H}, \mathsf{object}\ o:c\ \{f = u; \bar{V}\}, \mathsf{thread}\ p\ \{\ \mathsf{set}\ o.f = v; \vec{C}\ \} \twoheadrightarrow \bar{D} \vdash \bar{H}, \mathsf{object}\ o:c\ \{f = v; \bar{V}\}, \mathsf{thread}\ p\ \{\ \vec{C}\ \}}\ (\mathrm{R_C\text{-}SET})$$

Table 3 Class-Based Typing (Names)

$\boxed{\bar{D} \vdash c <: d}$

(S-CLASS)
$$\frac{\bar{D} \ni \mathsf{class}\ c <: d\ \{\_\} \quad \bar{D} \vdash d\ :\ \mathsf{Class}}{\bar{D} \vdash c <: d}$$

(S-REFLEX)
$$\frac{\bar{D} \vdash t\ :\ \mathsf{Class}}{\bar{D} \vdash t <: t}$$

(S-TRANS)
$$\frac{\bar{D} \vdash t <: s \quad \bar{D} \vdash s <: r}{\bar{D} \vdash t <: r}$$

$\boxed{\bar{D};\ \bar{Z} \vdash v\ :\ t}$

(T_C-NAME-VALUE)
$$\frac{\bar{Z} \ni v : s \quad \bar{D} \vdash s <: t}{\bar{D};\ \bar{Z} \vdash v\ :\ t}$$

$\boxed{\bar{D} \vdash c\ :\ \mathsf{Class}}$

(T_C-NAME-CLASS-OBJECT)
$$\frac{}{\bar{D} \vdash \mathsf{Object}\ :\ \mathsf{Class}}$$

(T_C-NAME-CLASS-CLASS)
$$\frac{\bar{D} \ni \mathsf{class}\ c <: d\ \{\_\} \quad \bar{D} \vdash d\ :\ \mathsf{Class}}{\bar{D} \vdash c\ :\ \mathsf{Class}}$$

$\boxed{\bar{D} \vdash m\ :\ \mathsf{Mth}(\vec{t}) : r\ \mathsf{for}\ s\ \mathsf{in}\ c}$

(T_C-NAME-METHOD)
$$\frac{\bar{D} \ni \mathsf{class}\ d <: \_\ \{\_\ \bar{M}\} \quad \bar{M} \ni \mathsf{protected}\ s\ \mathsf{method}\ m(\_ : \vec{t}) : r\ \{\_\} \quad \bar{D} \vdash c <: d}{\bar{D} \vdash m\ :\ \mathsf{Mth}(\vec{t}) : r\ \mathsf{for}\ s\ \mathsf{in}\ c}$$

$\boxed{\bar{D} \vdash f\ :\ \mathsf{Fld}\ r\ \mathsf{for}\ s\ \mathsf{in}\ c}$

(T_C-NAME-FIELD)
$$\frac{\bar{D} \ni \mathsf{class}\ d <: \_\ \{\bar{F}\ \_\} \quad \bar{F} \ni \mathsf{protected}\ s\ \mathsf{field}\ f : r\ ; \quad \bar{D} \vdash c <: d}{\bar{D} \vdash f\ :\ \mathsf{Fld}\ r\ \mathsf{for}\ s\ \mathsf{in}\ c}$$

of *bound name* for method declarations and command sequences. The class declaration "new class $c <: d\ \{\bar{F}\ \bar{M}\}; \vec{C}$" binds c, with scope $\bar{F}\ \bar{M}$ and $\vec{C}$. The declaration protected s method $m(\vec{X}) : r\ \{\vec{C}\}$ binds $\vec{X}$ and this, the scope is $\vec{C}$. Each let-command sequence "let $x = \ldots; \vec{C}$", binds x, with scope $\vec{C}$. The object declaration "new object $o : c\ \{\bar{V}\}; \vec{C}$" binds o, with scope $\bar{V}$ and $\vec{C}$.

3.2 Dynamic semantics

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

As a step towards the later development of the aspect based calculus, we model stack frames in the operational semantics. In a pushed stack frame "let $X = o\{S\}; \vec{C}$", we often say that o is the controlling object of S. A pushed frame "let $X = o\{S\}; \vec{C}$" successfully terminates in a return command which removes the remainder of S, leaving $\vec{C}$ to execute; "let $x = p\{\mathsf{return}\ v; \vec{B}\}; \vec{C}$" reduces to "let $x = v; \vec{C}$", which is then further reduced via substitution to "$\vec{C}[v/x]$".

The reduction rules are given in Table 2. The rules (R_C-LET) and (R_C-RETURN) deal with pushed frames. The rule (R_C-VALUE) allows returned values to be substituted through for the variables to which they are bound. The rules (R_C-GET) and (R_C-SET) allow for the manipulation of fields. The rule (R_C-NEW) allow threads to create new

Table 4 Class-Based Typing (Declarations)

$\boxed{\vdash P \ : \ \mathsf{Prog}}$

$(\text{T}_\text{C}\text{-PROG})$
$$\frac{\begin{array}{l}\overline{D};\ \overline{Z} \vdash \overline{H} \ : \ \mathsf{Heap} \\ \overline{D};\ \overline{Z} \vdash \overline{D} \ : \ \mathsf{Dec} \\ \mathsf{objects}(\overline{H}) = \overline{Z} \\ \text{domains } \overline{D},\ \overline{Z} \text{ disjoint}\end{array}}{\vdash (\overline{D} \vdash \overline{H}) \ : \ \mathsf{Prog}}$$

$\boxed{\overline{D};\ \overline{Z} \vdash E \ : \ \mathsf{Dec}}$

$(\text{T}_\text{C}\text{-DEC-CLASS})$
$$\frac{\begin{array}{l}\quad \overline{D} \vdash \overline{F} \ : \ \mathsf{Fld} \\ \overline{D};\ \overline{Z} \vdash \overline{M} \ : \ \mathsf{Mth\ in}\ c \\ \quad \overline{D} \vdash c \text{ is sane}\end{array}}{\overline{D};\ \overline{Z} \vdash \mathsf{class}\ c <: d\ \{\overline{F}\ \overline{M}\} \ : \ \mathsf{Dec}}$$

$\boxed{\overline{D} \vdash c \text{ is sane}}$

$(\text{T}_\text{C}\text{-SANE})$
$$\frac{\begin{array}{l}\overline{D} \vdash c <: \mathsf{Object} \\ c \neq \mathsf{Object} \\ \text{if } \overline{D} \vdash m \ : \ \mathsf{Mth}(\vec{t}_1) : r_1 \ \mathsf{for}\ s_1 \ \mathsf{in}\ c \\ \quad \text{and } \overline{D} \vdash m \ : \ \mathsf{Mth}(\vec{t}_2) : r_2 \ \mathsf{for}\ s_2 \ \mathsf{in}\ c \\ \quad \text{then } (\vec{t}_1, r_1, s_1) = (\vec{t}_2, r_2, s_2) \\ \text{if } \overline{D} \vdash f \ : \ \mathsf{Fld}\ r_1 \ \mathsf{for}\ s_1 \ \mathsf{in}\ c \\ \quad \text{and } \overline{D} \vdash f \ : \ \mathsf{Fld}\ r_2 \ \mathsf{for}\ s_2 \ \mathsf{in}\ c \\ \quad \text{then } (r_1, s_1) = (r_2, s_2)\end{array}}{\overline{D} \vdash c \text{ is sane}}$$

$\boxed{\mathsf{objects}(\overline{H}) = \overline{Z}}$

$(\text{T}_\text{C}\text{-OBJS})$
$$\frac{}{\mathsf{objects}(\overline{H}) = \{o : c \mid \mathsf{object}\ o : c\ \{\_\} \in \overline{H}\}}$$

$\boxed{\overline{D};\ \overline{Z} \vdash M \ : \ \mathsf{Mth\ in}\ c}$

$(\text{T}_\text{C}\text{-METHOD})$
$$\frac{\begin{array}{l}\overline{D} \vdash s, \vec{t}, r \ : \ \mathsf{Class} \\ \overline{D};\ \overline{Z}, \vec{x} : \vec{t}, \mathsf{this} : c \vdash \vec{C} \ : \ \mathsf{Stk}\ r\ \mathsf{for}\ c\end{array}}{\overline{D};\ \overline{Z} \vdash \mathsf{protected}\ s\ \mathsf{method}\ m(\vec{x} : \vec{t}) : r\ \{\vec{C}\} \ : \ \mathsf{Mth\ in}\ c}$$

$\boxed{\overline{D} \vdash F \ : \ \mathsf{Fld}}$

$(\text{T}_\text{C}\text{-FIELD})$
$$\frac{\overline{D} \vdash s, t \ : \ \mathsf{Class}}{\overline{D} \vdash \mathsf{protected}\ s\ \mathsf{field}\ f : t; \ : \ \mathsf{Fld}}$$

$\boxed{\overline{D};\ \overline{Z} \vdash H \ : \ \mathsf{Heap}}$

$(\text{T}_\text{C}\text{-HEAP-THREAD})$
$$\frac{\begin{array}{l}\overline{D};\ \overline{Z} \vdash S \ : \ \mathsf{Stk}\ \_\ \mathsf{for}\ c \\ \overline{D};\ \overline{Z} \vdash o \ : \ c\end{array}}{\overline{D};\ \overline{Z} \vdash \mathsf{thread}\ o\ \{S\} \ : \ \mathsf{Heap}}$$

$(\text{T}_\text{C}\text{-HEAP-OBJECT})$
$$\frac{\begin{array}{l}\overline{D} \vdash c \ : \ \mathsf{Class} \\ \overline{D} \vdash \overline{f} \ : \ \mathsf{Fld}\ \overline{t}\ \mathsf{for}\ \_\ \mathsf{in}\ c \\ \overline{D};\ \overline{Z} \vdash \overline{v} : \overline{t} \\ \forall f.\ \overline{D} \vdash f \ : \ \mathsf{Fld}\ t\ \mathsf{for}\ \_\ \mathsf{in}\ c\ \mathsf{implies}\ f \in \overline{f}\end{array}}{\overline{D};\ \overline{Z} \vdash \mathsf{object}\ o : c\ \{\overline{f = v};\ \} \ : \ \mathsf{Heap}}$$

classes, objects and threads. Note that we rely on alpha-equivalence to generate the names of references.

The rules $(\text{R}_\text{C}\text{-DYN-MSG})$ and $(\text{R}_\text{C}\text{-STC-MSG})$ 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 $c :: m$.

3.3 Static semantics

The type system for the class-based language is standard for the features being handled: mutable state, inner classes and concurrency. Whereas concurrency presents no additional difficulty over the sequential case, mutability and inner classes do impose upon the formalities somewhat. We formalize inner classes by allowing for new class declarations to appear in threads. We do not address issues of genericity. In our type system, field and method types are invariant under subclassing.

We refine the usual types in OO programs by including assertions on the type of the controlling object in anticipation of their use in the typing of the aspect based calculus. For example, the form of judgments on methods is:

$$\overline{D} \vdash m \ : \ \mathsf{Mth}(\vec{t}) : r\ \mathsf{for}\ s\ \mathsf{in}\ c$$

Table 5 Class-Based Typing (Commands)

$$\boxed{\bar{D};\ \bar{Z} \vdash S\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c}$$

($\mathrm{T_C}$-STK-LET)
$$\frac{\bar{D};\ \bar{Z} \vdash p\ :\ d \quad \bar{D};\ \bar{Z} \vdash S\ :\ \mathsf{Stk}\ t\ \mathsf{for}\ d \quad \bar{D};\ \bar{Z}, x{:}t \vdash \vec{C}\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c}{\bar{D};\ \bar{Z} \vdash \mathsf{let}\ x{:}t{=}p\{S\};\vec{C}\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c}$$

($\mathrm{T_C}$-STK-RETURN)
$$\frac{\bar{D};\ \bar{Z} \vdash v\ :\ r}{\bar{D};\ \bar{Z} \vdash \mathsf{return}\ v;\vec{C}\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c}$$

($\mathrm{T_C}$-STK-VALUE)
$$\frac{\bar{D};\ \bar{Z} \vdash v\ :\ t \quad \bar{D};\ \bar{Z}, x{:}t \vdash \vec{C}\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c}{\bar{D};\ \bar{Z} \vdash \mathsf{let}\ x{:}t{=}v;\vec{C}\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c}$$

($\mathrm{T_C}$-STK-DYN-MSG)
$$\frac{\bar{D};\ \bar{Z} \vdash o\ :\ d \quad \bar{D};\ \bar{Z} \vdash \mathsf{let}\ x{:}t{=}o.d{::}m(\vec{v})\ ;\vec{C}\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c}{\bar{D};\ \bar{Z} \vdash \mathsf{let}\ x{:}t{=}o.m(\vec{v})\ ;\vec{C}\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c}$$

($\mathrm{T_C}$-STK-STC-MSG)
$$\frac{\begin{array}{l}\bar{D};\ \bar{Z} \vdash o\ :\ d \quad \bar{D} \vdash m\ :\ \mathsf{Mth}(\vec{u}){:}t'\ \mathsf{for}\ s\ \mathsf{in}\ d \\ \bar{D};\ \bar{Z} \vdash \vec{v}\ :\ \vec{u} \quad \bar{D} \vdash t' <: t \quad \bar{D} \vdash c <: s \\ \bar{D};\ \bar{Z}, x{:}t \vdash \vec{C}\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c\end{array}}{\bar{D};\ \bar{Z} \vdash \mathsf{let}\ x{:}t{=}o.d{::}m(\vec{v})\ ;\vec{C}\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c}$$

($\mathrm{T_C}$-STK-GET)
$$\frac{\begin{array}{l}\bar{D};\ \bar{Z} \vdash o\ :\ d \quad \bar{D} \vdash f\ :\ \mathsf{Fld}\ t'\ \mathsf{for}\ s\ \mathsf{in}\ d \\ \bar{D} \vdash t' <: t \quad \bar{D} \vdash c <: s \\ \bar{D};\ \bar{Z}, x{:}t \vdash \vec{C}\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c\end{array}}{\bar{D};\ \bar{Z} \vdash \mathsf{let}\ x{:}t{=}o.f\ ;\vec{C}\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c}$$

($\mathrm{T_C}$-STK-SET)
$$\frac{\begin{array}{l}\bar{D};\ \bar{Z} \vdash o\ :\ d \quad \bar{D} \vdash f\ :\ \mathsf{Fld}\ t\ \mathsf{for}\ s\ \mathsf{in}\ d \\ \bar{D};\ \bar{Z} \vdash v\ :\ t \quad \bar{D} \vdash c <: s \\ \bar{D};\ \bar{Z} \vdash \vec{C}\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c\end{array}}{\bar{D};\ \bar{Z} \vdash \mathsf{set}\ o.f{=}v;\vec{C}\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c}$$

($\mathrm{T_C}$-STK-DEC)
$$\frac{\begin{array}{l}\bar{D}, \bar{E};\ \bar{Z}, \bar{Y} \vdash \bar{E}\ :\ \mathsf{Dec} \\ \bar{D}, \bar{E};\ \bar{Z}, \bar{Y} \vdash \bar{G}\ :\ \mathsf{Heap} \\ \bar{D}, \bar{E};\ \bar{Z}, \bar{Y} \vdash \vec{C}\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c \\ \mathrm{objects}(\bar{G}) = \bar{Y} \\ \mathrm{domains}\ \bar{D}, \bar{E}, \bar{Z}, \bar{Y}\ \mathrm{disjoint}\end{array}}{\bar{D};\ \bar{Z} \vdash \mathsf{new}\ \bar{E}\ \bar{G};\vec{C}\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c}$$

These types indicate that the result type is a subclass of r, the caller is constrained to be a subtype of s and the method itself is in the class c. ($\mathrm{T_C}$-NAME-METHOD) in Table 3 ties up this typing judgments with the method declaration protected s method $m(\vec{X}){:}r\,\{\vec{C}\}$.

Similarly, the form of judgment on fields is:

$$\bar{D} \vdash f\ :\ \mathsf{Fld}\ r\ \mathsf{for}\ s\ \mathsf{in}\ c$$

($\mathrm{T_C}$-NAME-FIELD) in Table 3 ties up this typing judgments with the field declaration protected s field $f{:}t;$.

Table 3 also describes the lookup of types of object names in the typing environment $\bar{Z}$, and ensures that Object is a valid class, and that valid classes are closed under subclassing.

The typing rules for declarations are described in Table 4.These are mostly standard. Note, however, that our calculus excludes class methods and constructors. We therefore allow objects references to occur in class declarations, as formalized in ($\mathrm{T_C}$-PROG). The

rule (T_C-SANE) ensures that subclasses are *invariant* with respect to the types of fields and the argument and return types of methods.

The judgments for commands are of the form:

$$\bar{D}; \bar{Z} \vdash S : \mathsf{Stk}\ r\ \mathsf{for}\ c$$

where r is read as the type of return values and c is read as the type of the class in whose control this command is being executed. Thus, from (T_C-METHOD) we deduce that a method declaration is valid in class c if the method body is typable as above in a type environment with this having type c.

Keeping this intuition in mind, the typing of commands is standard, and is described in Table 5. The usual case of binding values to let variables as described in (T_C-STK-VALUE). The case of nested stack frames, (T_C-STK-LET), is essentially similar from a typing point of view. (T_C-STK-RETURN) embodies the idea that returns terminate stack frames. The imperative features are handles as usual by (T_C-STK-GET) and (T_C-STK-SET).

Following the dynamic semantics, (T_C-STK-DYN-MSG) reduces the typing of a dynamic message dispatch to a static message dispatch. The case for static message dispatch is the sole place in the typing rules for commands that one sees the effect of carrying around type information of the controlling object. The first four premises of this rule are standard, enforcing the typing of the object name and the correct match between the type of the method, the types of the arguments and the return type. The hypothesis $\bar{D} \vdash c <: s$ ensures that the controlling context c has required access to the method m.

We now state progress and preservation properties; proofs can be found in Appendix A. We begin by identifying threads that have terminated normally.

DEFINITION 1 (TERMINATED THREAD). A thread declaration $\mathsf{thread}\ p\ \{\vec{C}\}$ has *terminated* if $\vec{C}$ is of the form $\mathsf{return}\ v; \vec{C}'$.

THEOREM 1 (CBL PROGRESS).
If $\vdash (\bar{D} \vdash \bar{H}, \mathsf{thread}\ p\ \{S\}) : \mathsf{Prog}$ *then either* $\mathsf{thread}\ p\ \{S\}$ *has terminated or:*

$$(\bar{D} \vdash \bar{H}, \mathsf{thread}\ p\ \{S\}) \rightarrow (\bar{D}' \vdash \bar{H}', \mathsf{thread}\ p\ \{S'\})$$

THEOREM 2 (CBL SUBJECT REDUCTION).
If $\vdash P : \mathsf{Prog}$ *and* $P \rightarrow^* P'$ *then* $\vdash P : \mathsf{Prog}$.

4 An Aspect-based Language

4.1 Syntax

We will 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 $\bar{a}$.

The move from a class-based language to an aspect-based language involves three new pieces of syntax: advice declarations, advised method calls and proceed calls. In Table 6 we extend the grammar for declarations and commands, replace the grammar for method declarations, and define a new grammar for pointcuts.

Table 6 Aspect-Based Syntax

Extend Table 1, modifying the syntax for method values and dynamic messages.

$\rho ::=$	*Placement*	$\phi, \psi ::=$	*Pointcut*		
around	Around	$\mathsf{call}(L)$	Call		
replace	Replace	$\mathsf{exec}(L)$	Execution		
$D, E ::= \ ...$	*Declaration*	true	True		
ρ advice $a(\vec{X}) : r$ at $\phi \{ \vec{C} \}$	Advice	false	False		
		$\neg\phi$	Negation		
$M ::=$ protected s method $m(\vec{t}) : r \ [\overline{a} ; \overline{b}]$	*Method*	$\phi \wedge \psi$	Conjunction		
$L ::= c :: m$	*Label*	$\phi \vee \psi$	Disjunction		
$C, B ::= \ ...$	*Command*				
let $X = o : c.m(\vec{v})$;	Dynamic Msg				
let $X = o : c.m[\overline{a} ; \overline{b}](\vec{v})$;	Advised Msg				
let $X =$ proceed $(\vec{v})$;	Proceed				

An advice declaration, new ρ advice $a(\vec{X}) : r$ at $\phi \{ \vec{C} \} ; \vec{B}$ has four essential components. The name a allows references to the aspect from elsewhere in the program. The command sequence $\vec{C}$ specifies *what* to execute and the pointcut ϕ specifies *when*. ρ is replace only if there are no occurrences of proceed in $\vec{C}$. The advice declaration binds a, with scope $\vec{B}$ and $\vec{C}$, and also binds $\vec{X}$, this and target, with scope $\vec{B}$.

A pointcut specifies the set of methods that are affected by this advice; formally pointcuts are presented as elements of the boolean algebra whose atoms are execution pointcuts and call pointcuts. Point cuts apply not only to the specified class, but to all subclasses as well; negation is useful to control this.

In the aspect language, class declarations contain methods of the form:

$$\mathsf{class}\ c <: d \ \{ \ ... \ \mathsf{protected}\ s\ \mathsf{method}\ m(\vec{u}) : r \ [\overline{a} ; \overline{b}] \ ... \ \}$$

The method declaration no longer includes a command sequence, but rather two sets of advice; $\overline{a}$ is executed by the *caller* (call advice), $\overline{b}$ is executed by the *callee* (execution advice). Method bodies in class declarations are redundant [15].

Advised method calls are not required in source programs; rather, they arise naturally during the dynamics. An advised method call "$o : c.m[\overline{a} ; \overline{b}](\vec{v})$;" indicates the collections of call advice $\overline{a}$ and execution advice $\overline{b}$ yet to be performed. 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, each dynamically dispatched method call must be annotated with a static type c.

The encoding of the class-based language into the aspect calculus provides some preliminary 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 "cbl_c_m" to identify a fresh name generated from class name c and method name m. Given a method definition

$$\mathsf{class}\ d <: c \ \{ \ ... \ \mathsf{protected}\ s\ \mathsf{method}\ m(\vec{x} : \vec{t}) : r \ \{ \vec{C} \} \ ... \ \}$$

Table 7 Aspect-Based Reduction

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

$$\frac{}{\boxed{\overline{D} \Vdash \mathsf{advice}(c :: m) = [\overline{a}\,;\,\overline{b}]}}\ (L_A\text{-TOP})$$
$$\overline{D} \Vdash \mathsf{advice}(\mathsf{Object} :: m) = [\emptyset\,;\,\emptyset]$$

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

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

$$\frac{\mathsf{domains}\ \overline{D}, \overline{E}, \overline{H}, \overline{G}\ \mathsf{disjoint}}{\begin{array}{l}\overline{D} \vdash \overline{H}, \mathsf{thread}\ p\ \{\ \mathsf{new}\ \overline{E}\ \overline{G}\,;\vec{C}\ \} \\ \quad \rightarrow\!\triangleright\ \mathsf{close}(\overline{D}, \overline{E}) \vdash \overline{H}, \overline{G}, \mathsf{thread}\ p\ \{\ \vec{C}\ \}\end{array}}\ (R_A\text{-NEW})\quad \boxed{\overline{D} \vdash \overline{H}, \mathsf{thread}\ p\ \{\ S\ \} \rightarrow\!\triangleright\ \overline{D}' \vdash \overline{H}', \mathsf{thread}\ p\ \{\ S'\ \}}$$

$$\frac{\overline{H} \ni \mathsf{object}\ o : d\ \{\ \_\ \} \quad \overline{D} \Vdash \mathsf{advice}(c :: m) = [\overline{a}\,;\,\_] \quad \overline{D} \Vdash \mathsf{advice}(d :: m) = [\_\,;\,\overline{b}]}{\begin{array}{l}\overline{D} \vdash \overline{H}, \mathsf{thread}\ p\ \{\ \mathsf{let}\ X = o : c.m(\vec{v})\,;\vec{C}\ \} \\ \quad \rightarrow\!\triangleright\ \overline{D} \vdash \overline{H}, \mathsf{thread}\ p\ \{\ \mathsf{let}\ X = o : c.m[\overline{a}\,;\,\overline{b}]\ (\vec{v})\,;\vec{C}\ \}\end{array}}\ (R_A\text{-DYN-MSG})$$

$$\frac{\overline{D} \Vdash \mathsf{advice}(c :: m) = [\_\,;\,\overline{b}]}{\begin{array}{l}\overline{D} \vdash \overline{H}, \mathsf{thread}\ p\ \{\ \mathsf{let}\ X = o.c :: m(\vec{v})\,;\vec{C}\ \} \\ \quad \rightarrow\!\triangleright\ \overline{D} \vdash \overline{H}, \mathsf{thread}\ p\ \{\ \mathsf{let}\ X = o : c.m[\emptyset\,;\,\overline{b}]\ (\vec{v})\,;\vec{C}\ \}\end{array}}\ (R_A\text{-STC-MSG})$$

$$\frac{\overline{D} \ni \_\ \mathsf{advice}\ b(\vec{x}) : \_\ \mathsf{at}\ \_\ \{\ \vec{B}\ \} \quad \vec{B}' = \vec{B}[^{o:c.m[\emptyset\,;\,\overline{b}]}/_{\mathsf{proceed}}]}{\begin{array}{l}\overline{D} \vdash \overline{H}, \mathsf{thread}\ p\ \{\ \mathsf{let}\ x : r = o : c.m[\emptyset\,;\,b, \overline{b}]\ (\vec{v})\,;\vec{C}\ \} \\ \quad \rightarrow\!\triangleright\ \overline{D} \vdash \overline{H}, \mathsf{thread}\ p\ \{\ \mathsf{let}\ x : r = o\{\ \vec{B}'[^{o}/_{\mathsf{this}}, ^{o}/_{\mathsf{target}}, ^{\vec{v}}/_{\vec{x}}]\ \}\,;\vec{C}\ \}\end{array}}\ (R_A\text{-ADV-MSG1})$$

$$\frac{\overline{D} \ni \_\ \mathsf{advice}\ a(\vec{x}) : \_\ \mathsf{at}\ \_\ \{\ \vec{B}\ \} \quad \vec{B}' = \vec{B}[^{o:c.m[\overline{a}\,;\,\overline{b}]}/_{\mathsf{proceed}}]}{\begin{array}{l}\overline{D} \vdash \overline{H}, \mathsf{thread}\ p\ \{\ \mathsf{let}\ x : r = o : c.m[a, \overline{a}\,;\,\overline{b}]\ (\vec{v})\,;\vec{C}\ \} \\ \quad \rightarrow\!\triangleright\ \overline{D} \vdash \overline{H}, \mathsf{thread}\ p\ \{\ \mathsf{let}\ x : r = p\{\ \vec{B}'[^{p}/_{\mathsf{this}}, ^{o}/_{\mathsf{target}}, ^{\vec{v}}/_{\vec{x}}]\ \}\,;\vec{C}\ \}\end{array}}\ (R_A\text{-ADV-MSG2})$$

Table 8 Pointcut Semantics

$$\text{(PC-ENAME)}\ \frac{\bar{D} \Vdash s :: m \in \text{execadv}(a)}{\bar{D} \ni \rho\ \mathsf{advice}\ a(\_) : \_\ \mathsf{at}\ \phi\ \{\ \_\ \} \quad \bar{D} \Vdash s :: m \in \text{execadv}(\phi)}$$

$$\text{(PC-CNAME)}\ \frac{\bar{D} \Vdash s :: m \in \text{calladv}(a)}{\bar{D} \ni \rho\ \mathsf{advice}\ a(\_) : \_\ \mathsf{at}\ \phi\ \{\ \_\ \} \quad \bar{D} \Vdash s :: m \in \text{calladv}(\phi)}$$

$$\text{(PC-EXEC)}\ \frac{\bar{D} \Vdash s :: m \in \text{execadv}(\phi)}{\bar{D} \vdash s <: t}{\bar{D} \Vdash s :: m \in \text{execadv}(\mathsf{exec}(t :: m))}$$

$$\text{(PC-CALL)}\ \frac{\bar{D} \Vdash s :: m \in \text{calladv}(\phi)}{\bar{D} \vdash s <: t}{\bar{D} \Vdash s :: m \in \text{calladv}(\mathsf{call}(t :: m))}$$

$$\text{(PC-ENOT)}\ \frac{\bar{D} \Vdash L \notin \text{execadv}(\phi)}{\bar{D} \Vdash L \in \text{execadv}(\neg\phi)} \qquad \text{(PC-CNOT)}\ \frac{\bar{D} \Vdash L \notin \text{calladv}(\phi)}{\bar{D} \Vdash L \in \text{calladv}(\neg\phi)}$$

$$\text{(PC-EAND)}\ \frac{\bar{D} \Vdash L \in \text{execadv}(\phi) \quad \bar{D} \Vdash L \in \text{execadv}(\psi)}{\bar{D} \Vdash L \in \text{execadv}(\phi \wedge \psi)} \qquad \text{(PC-CAND)}\ \frac{\bar{D} \Vdash L \in \text{calladv}(\phi) \quad \bar{D} \Vdash L \in \text{calladv}(\psi)}{\bar{D} \Vdash L \in \text{calladv}(\phi \wedge \psi)}$$

$$\text{(PC-EORL)}\ \frac{\bar{D} \Vdash L \in \text{execadv}(\phi)}{\bar{D} \Vdash L \in \text{execadv}(\phi \vee \psi)} \qquad \text{(PC-CORL)}\ \frac{\bar{D} \Vdash L \in \text{calladv}(\phi)}{\bar{D} \Vdash L \in \text{calladv}(\phi \vee \psi)}$$

$$\text{(PC-EORR)}\ \frac{\bar{D} \Vdash L \in \text{execadv}(\psi)}{\bar{D} \Vdash L \in \text{execadv}(\phi \vee \psi)} \qquad \text{(PC-CORR)}\ \frac{\bar{D} \Vdash L \in \text{calladv}(\psi)}{\bar{D} \Vdash L \in \text{calladv}(\phi \vee \psi)}$$

create the advice:

$$\mathsf{class}\ d <: c\ \{\ \ldots\ \mathsf{protected}\ s\ \mathsf{method}\ m(\vec{x}:\vec{t}) : r\ \ [\emptyset\ ;\ \mathsf{cbl}\_d\_m]\ \ldots\ \}$$
$$\mathsf{replace\ advice}\ \mathsf{cbl}\_d\_m(\vec{x}:\vec{t}) : r\ \mathsf{at}\ \mathsf{exec}(d :: m)\ \{\vec{C}\}$$

In this encoding, we presume that calls to "super" have been encoded using statically dispatched messages. In the example above, "super.m" is written "this.$c :: m$".

4.2 Dynamic Semantics

We write "$\bar{D} \Vdash s :: m \in \text{execadv}(\phi)$" when pointcut ϕ applies to the execution of method m in class s, and similarly we write "$\bar{D} \Vdash s :: m \in \text{calladv}(\phi)$ for call pointcuts. The semantics of pointcuts is defined in Table 8. The definition relies on a notion of subtyping, given in the Table 3. These definitions ignore the advice sets declared by methods. The cases (PC-EXEC) and (PC-CALL) handle the cases for subtyping. The other cases describe the extension to the full boolean algebra.

The semantics of aspect programs is defined in Table 7. 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 arguably should be based on the class hierarchy alone. Rules (L_A-TOP), (L_A-THIS) and (L_A-SUPER) do this.

(R_A-DYN-MSG) looks up the call and execution advice at different types, the call advice at the static type and the execution advice at the true dynamic type. The rule (R_A-ADV-MSG2) describes the reduction of call advice. this is substituted by the controlling object of the context p, target is substituted by the callee o, and the unrolled body itself is executed under the control of p. The rule (R_A-ADV-MSG1) describes the reduction of execution advice. Both this and target are substituted by the callee o, and the unrolled body itself is executed under the control of o.

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. In a coherent set of declarations the syntactic form of each advised message contains all of the advice names that apply to that method, given the dynamic context. To maintain coherence, the rule for inner declarations (R_A-NEW) uses *close* to saturate the declaration set any dynamically loaded advice.

DEFINITION 2 (COHERENCE). $\overline{D}$ is *coherent* if
$\overline{D} \ni \mathsf{class}\ c <: \_\ \{ \ldots \mathsf{protected}\ \_\ \mathsf{method}\ m(\_) : \_\ [\overline{a}\ ;\ \overline{b}] \ldots \}$ implies:

$$a \in \overline{a} \text{ iff } \overline{D} \Vdash c :: m \in \text{calladv}(a) \text{ and } b \in \overline{b} \text{ iff } \overline{D} \Vdash c :: m \in \text{execadv}(b)$$

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

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

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

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

LEMMA 1 (CLOSE). *If* $\Vdash (\overline{D} \vdash \overline{H}) : \mathsf{Prog}$, *then* close($\overline{D}$) *is coherent.*

LEMMA 2 (REDUCTION PRESERVES COHERENCE). *If P is coherent and $P \rightarrowtriangle^* P'$ then P' is coherent.*

4.3 Static Semantics

The static semantics of the aspect based language is presented in Tables 9 and 10.

Table 9 Aspect-Based Typing (Names, Declarations, Commands)

$\boxed{\overline{D};\ \overline{Z} \Vdash E\ :\ \mathsf{Dec}}$

($\mathrm{T_A}$-DEC-ADVICE)

$$\frac{\overline{D} \Vdash \phi\ :\ \mathsf{Ptcut}(\vec{u}) : r \text{ for } s \text{ in } t \qquad \overline{D};\ \overline{Z}, \vec{x}:\vec{u}, \mathsf{this}:s, \mathsf{target}:t \Vdash \vec{C}\ :\ \mathsf{Stk}\ \rho\ (\vec{u}) : r \text{ for } s}{\overline{D};\ \overline{Z} \Vdash \rho \text{ advice } a(\vec{x}:\vec{u}) : r \text{ at } \phi\ \{\vec{C}\}\ :\ \mathsf{Dec}}$$

$\boxed{\overline{D};\ \overline{Z} \Vdash M\ :\ \mathsf{Mth}\ \mathsf{in}\ c}$

($\mathrm{T_A}$-METHOD)

$$\frac{\forall a \in \overline{a}.\ \overline{D} \Vdash c :: m \in \mathrm{calladv}(a) \qquad \forall b \in \overline{b}.\ \overline{D} \Vdash c :: m \in \mathrm{execadv}(b) \qquad \exists b \in \overline{b}.\ \overline{D} \ni \mathsf{replace\ advice}\ b(\_):\_\ \mathsf{at}\ \_\ \{\_\}}{\overline{D};\ \overline{Z} \Vdash \mathsf{protected}\ s\ \mathsf{method}\ m(\vec{u}) : r\ [\overline{a}\,;\, b]\ :\ \mathsf{Mth}\ \mathsf{in}\ c}$$

$\boxed{\overline{D};\ \overline{Z} \Vdash H\ :\ \mathsf{Heap}}$

($\mathrm{T_A}$-HEAP-THREAD)

$$\frac{\overline{D};\ \overline{Z} \Vdash o\ :\ c \qquad \overline{D};\ \overline{Z} \Vdash S\ :\ \mathsf{Stk\ replace}():\_\ \mathsf{for}\ c}{\overline{D};\ \overline{Z} \Vdash \mathsf{thread}\ o\ \{S\}\ :\ \mathsf{Heap}}$$

$\boxed{\overline{D};\ \overline{Z} \Vdash S\ :\ \mathsf{Stk}\ \rho\ (\vec{t}) : r \text{ for } c}$

($\mathrm{T_A}$-STK-DYN-MSG)

$$\frac{\overline{D};\ \overline{Z} \Vdash \mathsf{let}\ x:t = o.d :: m(\vec{v})\,;\vec{C}\ :\ \mathsf{Stk}\ \rho\ (\vec{t}) : r \text{ for } c}{\overline{D};\ \overline{Z} \Vdash \mathsf{let}\ x:t = o:d.m(\vec{v})\,;\vec{C}\ :\ \mathsf{Stk}\ \rho\ (\vec{t}) : r \text{ for } c}$$

($\mathrm{T_A}$-STK-ADV-MSG)

$$\frac{\overline{D} \vdash d' <: d \qquad \overline{D} \Vdash \mathrm{advice}(d :: m) = [\overline{a}'\,;\_] \qquad \overline{a} \subseteq \overline{a}' \qquad \overline{D} \Vdash \mathrm{advice}(d' :: m) = [\_\,;\, \overline{b}'] \qquad \overline{b} \subseteq \overline{b}' \qquad \exists b \in \overline{b}.\ \overline{D} \ni \mathsf{replace\ advice}\ b(\_):\_\ \mathsf{at}\ \_\ \{\_\} \qquad \overline{D};\ \overline{Z} \Vdash o\ :\ d' \qquad \overline{D};\ \overline{Z} \Vdash \mathsf{let}\ x:t = o.d :: m(\vec{v})\,;\vec{C}\ :\ \mathsf{Stk}\ \rho\ (\vec{t}) : r \text{ for } c}{\overline{D};\ \overline{Z} \Vdash \mathsf{let}\ x:t = o:d.m\,[\overline{a}\,;\, \overline{b}]\ (\vec{v})\,;\vec{C}\ :\ \mathsf{Stk}\ \rho\ (\vec{t}) : r \text{ for } c}$$

($\mathrm{T_A}$-STK-PROCEED)

$$\frac{\overline{D};\ \overline{Z} \Vdash \vec{v}\ :\ \vec{t} \qquad \overline{D} \vdash r <: t \qquad \overline{D};\ \overline{Z}, x:r \Vdash \vec{C}\ :\ \mathsf{Stk\ around}\ (\vec{t}) : r \text{ for } c}{\overline{D};\ \overline{Z} \Vdash \mathsf{let}\ x:t = \mathsf{proceed}\ (\vec{v})\,;\vec{C}\ :\ \mathsf{Stk\ around}\ (\vec{t}) : r \text{ for } c}$$

Table 10 Aspect-Based Typing (Pointcuts)

Use de Morgan duality to move negation to the atoms, i.e. call and exec pointcuts.

$$\boxed{\bar{D} \Vdash \phi \,:\, \mathsf{Ptcut}(\vec{u}){:}r \;\mathsf{for}\; s \;\mathsf{in}\; t}$$

$$\frac{\bar{D} \Vdash t, s, \vec{u}, r \,:\, \mathsf{Class}}{\bar{D} \Vdash \mathsf{false} \,:\, \mathsf{Ptcut}(\vec{u}){:}r \;\mathsf{for}\; s \;\mathsf{in}\; t} \quad (\mathrm{T_A\text{-}PC\text{-}FALSE})$$

$$\frac{\bar{D} \Vdash \phi \,:\, \mathsf{Ptcut}(\vec{u}){:}r \;\mathsf{for}\; s \;\mathsf{in}\; t \qquad \bar{D} \Vdash \psi \,:\, \mathsf{Ptcut}(\vec{u}){:}r \;\mathsf{for}\; s \;\mathsf{in}\; t}{\bar{D} \Vdash \phi \vee \psi \,:\, \mathsf{Ptcut}(\vec{u}){:}r \;\mathsf{for}\; s \;\mathsf{in}\; t} \quad (\mathrm{T_A\text{-}PC\text{-}OR})$$

$$\frac{\bar{D} \Vdash \phi \,:\, \mathsf{Ptcut}(\vec{u}){:}r \;\mathsf{for}\; s \;\mathsf{in}\; t}{\bar{D} \Vdash \phi \wedge \psi \,:\, \mathsf{Ptcut}(\vec{u}){:}r \;\mathsf{for}\; s \;\mathsf{in}\; t} \quad (\mathrm{T_A\text{-}PC\text{-}AND\text{-}1})$$

$$\frac{\bar{D} \Vdash \psi \,:\, \mathsf{Ptcut}(\vec{u}){:}r \;\mathsf{for}\; s \;\mathsf{in}\; t}{\bar{D} \Vdash \phi \wedge \psi \,:\, \mathsf{Ptcut}(\vec{u}){:}r \;\mathsf{for}\; s \;\mathsf{in}\; t} \quad (\mathrm{T_A\text{-}PC\text{-}AND\text{-}2})$$

$$\frac{\bar{D} \Vdash m \,:\, \mathsf{Mth}(\vec{u}){:}r \;\mathsf{for}\; s' \;\mathsf{in}\; c \qquad \bar{D} \vdash s' <: s \qquad \bar{D} \vdash c <: t}{\bar{D} \Vdash \mathsf{call}(c :: m) \,:\, \mathsf{Ptcut}(\vec{u}){:}r \;\mathsf{for}\; s \;\mathsf{in}\; t} \quad (\mathrm{T_A\text{-}PC\text{-}CALL})$$

$$\frac{\bar{D} \Vdash m \,:\, \mathsf{Mth}(\vec{u}){:}r \;\mathsf{for}\; \_ \;\mathsf{in}\; c \qquad \bar{D} \vdash c <: s \qquad \bar{D} \vdash c <: t}{\bar{D} \Vdash \mathsf{exec}(c :: m) \,:\, \mathsf{Ptcut}(\vec{u}){:}r \;\mathsf{for}\; s \;\mathsf{in}\; t} \quad (\mathrm{T_A\text{-}PC\text{-}EXEC})$$

We begin our description with the typing of pointcuts. We first use de Morgan duality to move negation to the atoms, i.e. call and exec pointcuts. These judgments are of the form:

$$\bar{D} \Vdash \phi \,:\, \mathsf{Ptcut}(\vec{u}){:}r \;\mathsf{for}\; s \;\mathsf{in}\; t$$

to indicate that the return type of the methods in the pointcut are subtypes of r, the caller type is to be a subtype of s and the method is in a class that is a subtype of t. These constraints should be viewed as applying to the body of advice associated with the pointcut. The rules are described in Table 10. The base cases are those corresponding to the pointcut for false, given by $(\mathrm{T_A}\text{-PC-FALSE})$, a pointcut that is never activated, and so imposes no restrictions whatsoever. The non-trivial base cases are those for the call and execution pointcuts, given by $(\mathrm{T_A}\text{-PC-CALL})$ and $(\mathrm{T_A}\text{-PC-EXEC})$ respectively. For a call pointcut, we require the types of the caller and callee are subtypes of the corresponding types in the method. In contrast, an execution pointcut is executed under the control of the callee object. So, in this case, the caller type information in the method is ignored. For a disjunction of two pointcuts, the body of the advice has to be correct for invocations from either disjunct. So, we have to enforce the restrictions from both disjuncts. Symmetric reasoning applies for the conjunction of two pointcuts. Note that there are no rules for negated atoms. A pointcut which involves negation is typechecked when it occurs as a conjunct, since the rule for conjunction only requires one of the branches to typecheck.

The typing of pointcuts is used in the typing of advice. In rule $(\mathrm{T_A}\text{-DEC-ADVICE})$ of Table 9, the typing judgment for advice is described with the following key features:

- The return type of the advice body and the pointcut match.
- The type of the controlling object in the advice body, the type of the this variable and the type of the caller object match.

– The type of the target variable and the type of the container of the method match.

We invite the reader to recall the substitutions for this, target variables for call and execution methods.

An advised method call is typable if its entire advice list is, and furthermore the advice list is guaranteed to terminate in a piece of advice that doesn't invoke proceed. This latter condition is captured by $\exists b \in \overline{b}.\ \overline{D} \ni \mathsf{replace\ advice}\ b(\_):\_\ \mathsf{at}\ \_\ \{\_\}$ in rule $(\mathrm{T_A}\text{-METHOD})$of Table 9. The invariant maintained is that an advice can be of form $\mathsf{replace\ advice}\ b(\_):\_\ \mathsf{at}\ \_\ \{\_\}$ only if it does not have a call to proceed, e.g. see $(\mathrm{T_A}\text{-STK-PROCEED})$ of Table 9. Similar intuitions underlie rule $(\mathrm{T_A}\text{-STK-ADV-MSG})$ of Table 9, where the advice list is forced to terminate in a piece of execution advice that does not have a call to proceed. The judgment $\overline{D};\ \overline{Z} \Vdash S : \mathsf{Stk}\ \rho(\vec{t}) : r\ \mathsf{for}\ c$ includes the rules from table 5, modified to include $\rho(\vec{t})$.

We outline proofs of progress and subject reduction in Appendix B.

THEOREM 3 (ABL PROGRESS).
If $\Vdash (\overline{D} \vdash \overline{H}, \mathsf{thread}\ p\ \{S\}) : \mathsf{Prog}$ *then either* $\mathsf{thread}\ p\ \{S\}$ *has terminated or:*

$$(\overline{D} \vdash \overline{H}, \mathsf{thread}\ p\ \{S\}) \twoheadrightarrow (\overline{D'} \vdash \overline{H'}, \mathsf{thread}\ p\ \{S'\})$$

THEOREM 4 (ABL SUBJECT REDUCTION).
If $\Vdash P : \mathsf{Prog}$ *and* $P \twoheadrightarrow^* P'$ *then* $\Vdash P : \mathsf{Prog}$.

5 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, and being essentially the one from our earlier work on the untyped calculus.

Clearly, programs where advice arrives dynamically without restrictions cannot be translated into the class-based language. However, static weaving can be achieved under the addition of the following premise to $(\mathrm{T_A}\text{-STK-DEC})$, which guarantees that new advice does not affect existing classes (it can only affect classes *s* and *t* which must be defined in $\overline{E}$).

$$\begin{array}{r} \text{if } \overline{E} \ni \rho\ \mathsf{advice}\ a(\vec{x}:\vec{u}):r\ \mathsf{at}\ \phi\ \{\vec{C}\} \\ \text{then } \overline{E} \Vdash \phi : \mathsf{Ptcut}(\vec{u}):r\ \mathsf{for}\ s\ \mathsf{in}\ t \end{array}$$

Moreover, we make a requirement of advised method calls, that the advice listed must be a suffix of the advice defined in the method (not just a subset). That is in $(\mathrm{T_A}\text{-STK-ADV-MSG})$ we add the requirements:

$$\overline{a'} = \_, \overline{a} \qquad \overline{b'} = \_, \overline{b}$$

We will call programs which typecheck in this strengthened type system *statically weavable*.

The definition of weaving is a restricted version of the algorithm discussed in our earlier work on the untyped aspect calculus, which we showed to respect the dynamic semantics: that is (up to some renaming of methods) we can complete the diagrams given in the introduction.

Fig. 1 Weaving Example

```
object p : Main { }
class Main {
  protected Main method main(Unit) : Unit [∅ ; ma]
}
replace advice ma(z : Unit) : Unit at exec(Main :: main) {
  let x = o : c.m(unit) ; return unit;
}
around advice ca(z : Unit) : Unit at call(c :: m) {
  let y = proceed(unit) ; return unit;
}
object o : c { }
class c {
  protected Main method m(Unit) : Unit [ca ; cb]
}
replace advice cb(z : Unit) : Unit at exec(c :: m) { return unit; }
```

```
object p : Main { }
class Main {
  protected Main method main(z : Unit) : Unit {
    skip; let x = this.call_ca_c_m(o) ; return unit;
  }
  protected Object method exec_ma(z : Unit) : Unit {
    skip; let x = this.call_ca_c_m(o) ; return unit;
  }
  protected Object method call_ca_c_m(z1 : c, z2 : Unit) : Unit {
    let y = z1.exec_cb() ; return unit;
  }
}
object o : c { }
class c {
  protected Main method m(z : Unit) : Unit { return unit; }
  protected Object method exec_cb(z : Unit) : Unit { return unit; }
}
```

For the indefatigable reader, we include the formal definition of weaving in appendix C. In this extended abstract, we discuss the basic ideas via an example, given in Figure 1. Weaving causes the creation of new methods. Execution advice causes the creation of new methods in the callee: thus, in the woven program, we have the new method exec_ma in Main, and exec_cb in class c. Call advice causes the creation of new methods in the caller: in the example, such a method is call_ca_c_m in Main. Note that this method has a parameter for the callee object. The effect of the proceed in advice ca is reflected in the method call on exec_cb in call_ca_c_m. To ensure precise correspondence of the reductions the code for methods is set to be that of the method corresponding to the first piece of execution advice: thus, method exec_ma and method

main in Main are identical, as are methods exec_cb and method m in c. Finally, the extra skip's are meant for bookkeeping to match up the reductions.

In this paper, we focus instead on the fact that weaving preserves typability.

THEOREM 5 (WEAVING PRESERVES TYPABILTY).
For any statically weavable P, if $\Vdash P : \mathsf{Prog}$ *and* $\mathrm{weave}(P) = P'$ *then* $\vdash P' : \mathsf{Prog}$.

6 Future work: The need for polymorphism.

The need for polymorphic types is motivated by an attempt at a translation of before advice into around advice. The intent of before advice in AspectJ is that it executes just before the method is called; so, one might be tempted to translate, as a first approximation, "`before(`$\vec{x}$`){C;}`" as "`around(`$\vec{x}$`){C; proceed(`$\vec{x}$`);}`". In a monomorphic typing scheme, such as the one presented in this paper, one needs a copy of this translation for every possible result type of *C*, and every possible type and length of the argument list.

It makes intuitive sense that the translation of before into around advice above works generically at all result types, since it is the *same translation* at all possible result types. This motivates the consideration of generic types [4, 13]. A related issue that needs to be addressed in this context is the issue of covariant subtyping on return values of methods. Consider the declarations

```
class c {
  protected Object method m():Object {
    return "Hello";
  }
}
class d <: c {
  method m():Integer {
    return new Integer(5);
  }
}
replace advice a():Object at exec(c::m) {
  return "World";
}
```

It is routine to turn this example into one which causes type safety to fail. In this example, advice is acting like method update [1], and so should be typed using invariant methods. AspectJ is based on Java 1.4 which uses invariant methods, and the AspectJ and Generic Java compilers are not currently compatible.

The translation of before advice into around advice motivates consideration of further kinds of polymorphism. For example, if the command *C* doesn't access the arguments $\vec{x}$, the translation of before advice into around advice is also polymorphic in the number and kinds of arguments. This is clearly in the spirit of row-polymorphism [23, 26].

In future work, we intend to explore a type system with polymorphism. On the one hand, such a type system will avoid the problems described above. On the other hand, such a type system can also aid a programmer by permitting the description of polymorphic pointcuts and advice. In terms of applications to language design, at the time of writing, there is no published proposal on how AspectJ will handle generic types and covariant method return types. However, the integration of aspects and generic types remains an active area of investigation (e.g. see FAQ on http://www.eclipse.org/aspectj/index.html).

7 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 [27], give a denotational semantics for a mini-language that embodies the key features of dynamic join points, pointcut designators, and advice. A suitable incorporation of these ideas into our work might enable our calculus to scale up into a model of a real-life aspect-oriented programming language.

The research closest to the spirit of our paper is the work of Krishnamurthi and Tucker [24] and Walker, Zdancewic and Ligatti [25]. Both these papers investigate the addition of aspects to a functional language paradigm. While Krishnamurthi and Tucker study both call and execution pointcuts, Walker, Zdancewic and Ligatti study only execution pointcuts. Walker, Zdancewic and Ligatti also study a type system on the target of the translation of the aspect oriented programs. In both cases, advice and join-points are first class entities that can be created and manipulated at runtime. In contrast to the ad-hoc handling of advice order in our paper, these papers elegantly model aspect scoping and order using the powerful mechanisms of the underlying functional paradigm. On the other hand, these papers do not study the issues of typing at the source level of class-based aspect programs in the presence of subtyping.

Summary

In this paper, we have developed a typed calculus of aspect programs. This calculus is expressive — it includes inner classes, concurrency and dynamic arrival of new advice. To our knowledge, this is the first source-level typing system for class-based aspect oriented programs.

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 *ECOOP Workshop on Object-Based Distributed Programming*, 1993.
3. 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.
4. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, 1998.
5. Kim B. Bruce. *Foundations of Object-Oriented Programming Languages: Types and Semantics*. MIT Press, 2002.
6. Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155:108–133, 1999.
7. 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)*, pages 104–127, 1997.
8. Kim B. Bruce, Adrian Fiech, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems*, 25(2), 2003.
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. The aspect sand box. 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 Symposium on Programming (ESOP)*, pages 135–149, 2000.
12. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 171–183, 1998.
13. Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.
14. Atsushi Igarashi and Benjamin C. Pierce. On inner classes. *Information and Computation*, 177(1):56–89, 2002.
15. Radha Jagadeesan, Alan Jeffrey, and James Riely. An untyped calculus of aspect oriented programs. In *European Conference on Object-Oriented Programming (ECOOP)*, 2003. To appear.
16. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 327–355, 2001.
17. 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)*, pages 220–242, 1997.
18. G. Leavens. Report on the foal 2002 workshop. http://www.cis.upenn.edu/~bcpierce/types/archives/current/msg01029.html%.
19. Gary T. Leavens and Ron Cytron, editors. *FOAL 2002 Proceedings*, 2002. Iowa State Univeristy Technical report 02-06, available from http://www.cs.iastate.edu/~leavens/FOAL/index-2002.html.

20. K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.
21. H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Symposium on Software Architectures and Component Technology*, pages 293–323, 2001.
22. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
23. Didier Rémy. Records and variants as a natural extension of ML. In *ACM Symposium on Principles Of Programming Languages (POPL)*, 1989.
24. David Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In *International Conference on Aspect Oriented Software Development (AOSD)*, 2003.
25. David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *ACM International Conference on Functional Programming (ICFP)*, 2003. To appear.
26. Mitchell Wand. Type inference for objects with instance variables and inheritance. In Carl Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 97–120. MIT Press, 1994.
27. Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *Workshop on Foundations of Object Oriented Languages (FOOL)*, pages 67–88, 2002.

A CBL Proofs

We now prove type preservation and progress results for the type system. The proofs follow the standard outline of such proofs in the literature: namely, weakening, strengthening, subsumption, substitutivity, and method lookup. A sketch of the formal presentation follows. Let $\mathcal{J}$ range over each of the judgments of the form $\overline{D};\ \overline{Z} \vdash \mathcal{J}$ or $\overline{D} \vdash \mathcal{J}$ in Tables 3,4 and 5.

New type assumptions preserve well-typedness.

LEMMA 3 (CBL WEAKENING).
If $\overline{D} \subseteq \overline{D'}$ *and* $\overline{Z} \subseteq \overline{Z'}$ *and* $\overline{D'}, \overline{Z'}$ *have disjoint domains, then* $\overline{D};\ \overline{Z} \vdash \mathcal{J}$ *implies* $\overline{D'};\ \overline{Z'} \vdash \mathcal{J}$.

The type assumption of a variable $x : t$ can be strengthened to $x : t'$, when $\overline{D} \vdash t' <: t$. The proof is by an induction on the derivation of $\overline{D};\ \overline{Z}, x : t \vdash \mathcal{J}$.

LEMMA 4 (CBL STRENGTHENING).
If $\overline{D} \vdash t' <: t$ *and* $\overline{D};\ \overline{Z}, x : t \vdash \mathcal{J}$ *then* $\overline{D};\ \overline{Z}, x : t' \vdash \mathcal{J}$.

In a command, the result type can be weakened and the type of controlling object can be strengthened. The proof is by an induction on the derivation of $\overline{D};\ \overline{Z} \vdash S\ :\ \mathsf{Stk}\ r'\ \mathsf{for}\ c$

LEMMA 5 (CBL STACK SUBSUMPTION).
If $\overline{D} \vdash c' <: c$ *and* $\overline{D} \vdash r' <: r$ *and* $\overline{D};\ \overline{Z} \vdash S\ :\ \mathsf{Stk}\ r'\ \mathsf{for}\ c$ *then* $\overline{D};\ \overline{Z} \vdash S\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c'$.

Variables can be substituted by well-typed values of the same type.

LEMMA 6 (CBL SUBSTITUTIVITY).
If $\overline{D};\ \overline{Z} \vdash v\ :\ t$ *and* $\overline{D};\ \overline{Z}, x : t \vdash \mathcal{J}$ *then* $\overline{D}[v/x];\ \overline{Z} \vdash \mathcal{J}[v/x]$.

In a well-typed method declaration, the body of the method typechecks with expected types under natural assumptions on the types of the this variable and the parameters.

LEMMA 7 (CBL METHOD LOOKUP).
If $\overline{D} \vdash m\ :\ \mathsf{Mth}(\vec{u}) : r\ \mathsf{for}\ s\ \mathsf{in}\ c$ *and* $\overline{D};\ \overline{Z} \vdash \overline{D}\ :\ \mathsf{Dec}$ *then* $\overline{D} \vdash \mathrm{mbody}(c :: m) = (\vec{x})\vec{C}$ *where* $\vec{x}$ *and* $\vec{u}$ *have the same length and* $\overline{D};\ \overline{Z}, \mathsf{this} : c, \vec{x} : \vec{u} \vdash \vec{C}\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ c$.

The proof of progress is standard given the above lemmas.

THEOREM (1; CBL PROGRESS).
If $\vdash (\overline{D} \vdash \overline{H}, \mathsf{thread}\ p\ \{S\})\ :\ \mathsf{Prog}$ *then either* $\mathsf{thread}\ p\ \{S\}$ *has terminated or:*

$$(\overline{D} \vdash \overline{H}, \mathsf{thread}\ p\ \{S\}) \rightarrow (\overline{D'} \vdash \overline{H'}, \mathsf{thread}\ p\ \{S'\})$$

Finally, we consider type preservation.

THEOREM (2; CBL SUBJECT REDUCTION).
If $\vdash P\ :\ \mathsf{Prog}$ *and* $P \rightarrow^* P'$ *then* $\vdash P\ :\ \mathsf{Prog}$.

Proof. We proceed by induction on the proof of $P \rightarrow^* P'$, where the interesting case is (R_C-STEP):

$$(\bar{D} \vdash \bar{H}, \mathsf{thread}\ p\ \{S\}) = P \rightarrow (\bar{D}' \vdash \bar{H}', \mathsf{thread}\ p\ \{S'\}) = P'$$

By Lemma 3, it suffices to show:

$$\begin{array}{ll} \bar{D}';\ \bar{Z}' \vdash \bar{D}' : \mathsf{Dec} & \bar{D}';\ \bar{Z}' \vdash \bar{H}' : \mathsf{Heap} \\ \bar{D}';\ \bar{Z}' \vdash S' : \mathsf{Stk}\ r\ \mathsf{for}\ c & \bar{D} \subseteq \bar{D}' \quad \bar{Z} \subseteq \bar{Z}' \\ \text{domains } \bar{D}', \bar{Z}' \text{ disjoint} & \text{objects}(\bar{H}') = \bar{Z}' \end{array}$$

In this extended abstract, we only consider the case of (R_C-DYN-MSG), corresponding to the novel feature in the type system. We have:

$$\begin{array}{ll} S = \mathsf{let}\ x{:}t{=}o.m(\vec{v})\ ;\vec{C} & \bar{H} \ni \mathsf{object}\ o{:}d'\ \{\_\} \\ S' = \mathsf{let}\ x{:}t{=}o\{\vec{B}[^{o}/_{\mathsf{this}}, ^{\vec{v}}/_{\vec{x}}]\}\ ;\vec{C} & \bar{D} \vdash \text{mbody}(d'::m) = (\vec{x})\vec{B} \\ \bar{D} = \bar{D}' & \bar{H} = \bar{H}' \end{array}$$

Since $\bar{D};\ \bar{Z} \vdash S : \mathsf{Stk}\ r\ \mathsf{for}\ c$ and $\text{objects}(\bar{H}) = \bar{Z}$ we have:

$$\begin{array}{ll} \bar{D};\ \bar{Z}, x{:}t \vdash \vec{C} : \mathsf{Stk}\ r\ \mathsf{for}\ c & \bar{D};\ \bar{Z} \vdash o : d \\ \bar{D} \vdash m : \mathsf{Mth}\,(\vec{u}){:}t'\ \mathsf{for}\ s\ \mathsf{in}\ d & \bar{D};\ \bar{Z} \vdash \vec{v} : \vec{u} \\ \bar{D} \vdash c <: s \qquad \bar{D} \vdash d' <: d & \bar{D} \vdash t' <: t \end{array}$$

so by Lemma 7 and Lemma 5 we have:

$$\bar{D};\ \bar{Z}, \vec{x}{:}\vec{u}, \mathsf{this}{:}d \vdash \vec{B} : \mathsf{Stk}\ t\ \mathsf{for}\ d$$

so by Lemma 6 we have: $\bar{D};\ \bar{Z} \vdash \vec{B}[^{o}/_{\mathsf{this}}, ^{\vec{v}}/_{\vec{x}}] : \mathsf{Stk}\ t\ \mathsf{for}\ d$ and so we use (R_C-LET) to get: $\bar{D};\ \bar{Z} \vdash S' : \mathsf{Stk}\ r\ \mathsf{for}\ c$ as required.

B ABL Proofs

The proofs for the aspect based language closely follow those for the class-based language: namely weakening, strengthening, subsumption and substitutivity. Instead of a treatment of method lookup, we have a treatment of advice lookup. The new features are the handling of pointcut typing, substitutivity of proceed, and the close() operation.

We begin with a formal statement of properties whose proofs are quite similar to the analogous properties of the CBL.

LEMMA 8 (ABL WEAKENING).
If $\bar{D} \subseteq \bar{D}'$ and $\bar{Z} \subseteq \bar{Z}'$ and $\bar{D}', \bar{Z}'$ have disjoint domains, then $\bar{D};\ \bar{Z} \Vdash \mathcal{J}$ implies $\bar{D}';\ \bar{Z}' \Vdash \mathcal{J}$.

LEMMA 9 (ABL STRENGTHENING).
If $\bar{D} \vdash t' <: t$ and $\bar{D};\ \bar{Z}, x{:}t \Vdash \mathcal{J}$ then $\bar{D};\ \bar{Z}, x{:}t' \Vdash \mathcal{J}$.

LEMMA 10 (ABL STACK SUBSUMPTION). *If* $\overline{D};\ \overline{Z} \Vdash S\ :\ \mathsf{Stk}\,\rho\,(\vec{u}) : r'$ for c *and* $\overline{D} \vdash c' <: c$ *and* $\overline{D} \vdash r' <: r$ *then* $\overline{D};\ \overline{Z} \Vdash S\ :\ \mathsf{Stk}\,\rho\,(\vec{u}) : r$ for c'.

LEMMA 11 (ABL SUBSTITUTIVITY).
If $\overline{D};\ \overline{Z} \Vdash v\ :\ t$ *and* $\overline{D};\ \overline{Z}, x : t \Vdash \mathcal{J}$ *then* $\overline{D}[v/x];\ \overline{Z} \Vdash \mathcal{J}[v/x]$.

An induction on the derivation of $\overline{D} \Vdash \phi\ :\ \mathsf{Ptcut}\,(\vec{u}) : r$ for s in t shows that methods and their associated pointcuts agree on types.

LEMMA 12 (ABL POINTCUT TYPING). *If* $\overline{D} \Vdash m\ :\ \mathsf{Mth}\,(\vec{u}') : r'$ for s' in t' *and* $\overline{D} \Vdash \phi\ :\ \mathsf{Ptcut}\,(\vec{u}) : r$ for s in t *then:*

1. $\overline{D} \Vdash t' :: m \in \mathrm{calladv}(\phi)$ *implies* $\vec{u}' = \vec{u}$ *and* $r' = r$ *and* $\overline{D} \vdash s' <: s$ *and* $\overline{D} \vdash t' <: t$.
2. $\overline{D} \Vdash t' :: m \in \mathrm{execadv}(\phi)$ *implies* $\vec{u}' = \vec{u}$ *and* $r' = r$ *and* $\overline{D} \vdash t' <: s$ *and* $\overline{D} \vdash t' <: t$.

An induction on the proof of $\overline{D} \Vdash \mathrm{advice}(c :: m) = [\overline{a}\,;\,\overline{b}]$, together with uses of Lemmas 9, 10 and 12 enables us to prove the advice analogue of the method lookup lemma 7.

LEMMA 13 (ABL ADVICE LOOKUP).
If $\overline{D} \Vdash m\ :\ \mathsf{Mth}\,(\vec{u}) : r$ for s in c *and* $\overline{D};\ \overline{Z} \Vdash \overline{D}\ :\ \mathsf{Dec}$ *then* $\overline{D} \Vdash \mathrm{advice}(c :: m) = [\overline{a}\,;\,\overline{b}]$ *where:*

1. *For any* $a \in \overline{a}$, *we have* $\overline{D} \ni \rho\ \mathsf{advice}\ a\,(\vec{x} : \vec{u}) : r\ \mathsf{at}\ \phi\ \{\,\vec{C}\,\}$ *where* $\overline{D};\ \overline{Z}, \vec{x} : \vec{u}, \mathsf{this} : s, \mathsf{target} : c \Vdash \vec{C}\ :\ \mathsf{Stk}\,\rho\,(\vec{u}) : r$ for s.
2. *For any* $b \in b$, *we have* $\overline{D} \ni \rho\ \mathsf{advice}\ b\,(\vec{x} : \vec{u}) : r\ \mathsf{at}\ \phi\ \{\,\vec{C}\,\}$ *where* $\overline{D};\ \overline{Z}, \vec{x} : \vec{u}, \mathsf{this} : c, \mathsf{target} : c \Vdash \vec{C}\ :\ \mathsf{Stk}\,\rho\,(\vec{u}) : r$ for c.

Inductions on the definition of $\mathrm{close}(\overline{E})$, and on the derivation of $\overline{D};\ \overline{Z} \Vdash \mathcal{J}$ show that the close() operation preserves well-typedness.

LEMMA 14 (ABL CLOSURE TYPING).

1. *If* $\overline{D};\ \overline{Z} \Vdash \overline{E}\ :\ \mathsf{Dec}$ *then* $\overline{D};\ \overline{Z} \Vdash \mathrm{close}(\overline{E})\ :\ \mathsf{Dec}$.
2. *If* $\overline{D};\ \overline{Z} \Vdash \mathcal{J}$ *then* $\mathrm{close}(\overline{D});\ \overline{Z} \Vdash \mathcal{J}$.

In the setting of the aspect calculus, we also have to show that the substitutions of the proceed variable in the dynamic semantics preserve typing. The proof is by an induction on the derivation of $\overline{D};\ \overline{Z} \Vdash \vec{B}\ :\ \mathsf{Stk}\,\rho\,(\vec{u}) : r$ for c, making use of Lemma 8.

LEMMA 15 (ABL PROCEED SUBSTITUTIVITY). *If* $\overline{D};\ \overline{Z} \Vdash o\ :\ d'$ *and* $\overline{D} \vdash c <: s$ *and* $\overline{D} \vdash d' <: d$ *and* $\overline{D} \Vdash m\ :\ \mathsf{Mth}\,(\vec{u}) : r$ for s in d *and* $\overline{D};\ \overline{Z} \Vdash \vec{B}\ :\ \mathsf{Stk}\,\rho\,(\vec{u}) : r$ for c *then:*

1. $\overline{D};\ \overline{Z} \Vdash \vec{B}[^{o.m}/_{\mathsf{proceed}}]\ :\ \mathsf{Stk}\ \mathsf{replace}\,() : r$ for c.
2. *If* $\overline{D} \Vdash \mathrm{advice}(d :: m) = [\overline{a}'\,;\,\_]$ *and* $\overline{a} \subseteq \overline{a}'$ *and* $\overline{D} \Vdash \mathrm{advice}(d' :: m) = [\_\,;\,\overline{b}']$ *and* $\overline{b} \subseteq \overline{b}'$ *and* $\exists b \in \overline{b}.\ \overline{D} \ni \mathsf{replace\ advice}\ b\,(\_) : \_\ \mathsf{at}\ \_\ \{\,\_\,\}$ *then* $\overline{D};\ \overline{Z} \Vdash \vec{B}[^{o : d.m\,[\overline{a}\,;\,\overline{b}]}/_{\mathsf{proceed}}]\ :\ \mathsf{Stk}\ \mathsf{replace}\,() : r$ for c.

As in the CBL, the proof of progress is by an induction on the derivation of:

$$\overline{D};\ \overline{Z} \Vdash S\ :\ \mathsf{Stk}\ \mathsf{replace}\,() : r\ \mathsf{for}\ c$$

THEOREM (3; ABL PROGRESS).
If $\Vdash (\bar{D} \vdash \bar{H}, \mathsf{thread}\ p\ \{S\})\ :\ \mathsf{Prog}$ *then either* $\mathsf{thread}\ p\ \{S\}$ *has terminated or:*

$$(\bar{D} \vdash \bar{H}, \mathsf{thread}\ p\ \{S\}) \rightarrowtriangle (\bar{D}' \vdash \bar{H}', \mathsf{thread}\ p\ \{S'\})$$

As in the CBL, the proof of type preservation is by induction on the proof of $P \rightarrowtriangle^* P'$.

THEOREM (4; ABL SUBJECT REDUCTION).
If $\Vdash P\ :\ \mathsf{Prog}$ *and* $P \rightarrowtriangle^* P'$ *then* $\Vdash P\ :\ \mathsf{Prog}$.

C Weaving

We give the definition of weaving for the typed language, starting with the rule for programs, then declarations, heap elements and commands. The last four rules are used for generating new method bodies.

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

$$\frac{}{\bar{D} \Vdash \mathsf{wdec}(\ldots \mathsf{advice} \ldots) = \emptyset}\ (\text{W-ADVICE}) \qquad \frac{\bar{D} \Vdash \mathsf{wmth}(c;\ \bar{M}) = \bar{M}'}{\begin{array}{l}\bar{D} \Vdash \mathsf{wdec}(\mathsf{class}\ c <: d\ \{\bar{F}\ \bar{M}\}) \\ \quad = \mathsf{class}\ c <: d\ \{\bar{F}\ \bar{M}'\}\end{array}}\ (\text{W-CLASS})$$

$$\frac{\begin{array}{l}\bar{D} \Vdash \mathsf{genExecMth}(c :: m;\ \bar{b}) = \bar{M} \\ \bar{M} \ni \mathsf{protected\ Object\ method\ exec\_}\bar{b}(\vec{x} : \vec{u}) : r\ \{\vec{C}\}\end{array}}{\begin{array}{l}\bar{D} \Vdash \mathsf{wmth}(c;\ \mathsf{protected}\ s\ \mathsf{method}\ m(\vec{u}) : r\ [\bar{a}\ ;\ b]) \\ \quad = \bar{M}, \mathsf{protected}\ s\ \mathsf{method}\ m(\vec{x} : \vec{u}) : r\ \{\vec{C}\}\end{array}}\ (\text{W-METHOD})$$

$$\frac{}{\bar{D} \Vdash \mathsf{wheap}(\mathsf{object}\ o : c\ \{\bar{V}\}) = \mathsf{object}\ o : c\ \{\bar{V}\}}\ (\text{W-OBJECT}) \qquad \frac{\bar{D} \Vdash \mathsf{wstack}(p\{S\}) = (\emptyset\ ;\ S')}{\bar{D} \Vdash \mathsf{wheap}(\mathsf{thread}\ p\ \{S\}) = \mathsf{thread}\ p\ \{S'\}}\ (\text{W-THREAD})$$

$$\frac{\begin{array}{l}\mathsf{close}(\bar{D}, \bar{E}) \Vdash \mathsf{wdec}(\mathsf{close}(\bar{E})) = \bar{E}' \\ \mathsf{close}(\bar{D}, \bar{E}) \Vdash \mathsf{wheap}(\bar{H}) = H' \\ \mathsf{close}(\bar{D}, \bar{E}) \Vdash \mathsf{wstack}(p\{\vec{C}\}) = (\bar{M}\ ;\ \vec{C}')\end{array}}{\begin{array}{l}\bar{D} \Vdash \mathsf{wstack}(p\{\mathsf{new}\ \bar{E}\ \bar{H}\ ;\ \vec{C}\}) \\ \quad = (\bar{M}\ ;\ \mathsf{new}\ \bar{E}'\ \bar{H}'\ ;\ \vec{C}')\end{array}}\ (\text{W-DEC}) \qquad \frac{\begin{array}{l}\bar{D} \Vdash \mathsf{wstack}(q\{S\}) = (\emptyset\ ;\ S') \\ \bar{D} \Vdash \mathsf{wstack}(p\{\vec{C}\}) = (\bar{M}\ ;\ \vec{C}')\end{array}}{\begin{array}{l}\bar{D} \Vdash \mathsf{wstack}(p\{\mathsf{let}\ X = q\{S\}\ ;\ \vec{C}\}) \\ \quad = (\bar{M}\ ;\ \mathsf{let}\ X = q\{S'\}\ ;\ \vec{C}')\end{array}}\ (\text{W-LET})$$

(W-DYN-MSG1)
$$\frac{\overline{D} \Vdash \mathsf{advice}(c::m) = [\emptyset\,;\,\_] \qquad \overline{D} \Vdash \mathsf{wstack}(p\{\,\vec{C}\,\}) = (\overline{M}\,;\,\vec{C}')}{\overline{D} \Vdash \mathsf{wstack}(p\{\,\mathsf{let}\,X = o : c.m(\vec{v})\,;\vec{C}\,\}) = (\overline{M}\,;\,\mathsf{skip}\,;\mathsf{let}\,X = o.m(\vec{v})\,;\vec{C}')}$$

(W-ADV-MSG1)
$$\frac{\overline{D} \Vdash \mathsf{wstack}(p\{\,\vec{C}\,\}) = (\overline{M}\,;\,\vec{C}')}{\overline{D} \Vdash \mathsf{wstack}(p\{\,\mathsf{let}\,X = o : c.m\,[\emptyset\,;\,\overline{b}]\,(\vec{v})\,;\vec{C}\,\}) = (\overline{M}\,;\,\mathsf{let}\,X = o.\mathsf{exec\_}\overline{b}(\vec{v})\,;\vec{C}')}$$

(W-ADV-MSG2)
$$\frac{\overline{D} \Vdash \mathsf{genCallMth}(c::m;\,\overline{a}) = \overline{M} \qquad \overline{D} \Vdash \mathsf{wstack}(p\{\,\vec{C}\,\}) = (\overline{M}'\,;\,\vec{C}')}{\overline{D} \Vdash \mathsf{wstack}(p\{\,\mathsf{let}\,X = o : c.m\,[\overline{a}\,;\,\_]\,(\vec{v})\,;\vec{C}\,\}) = (\overline{M}, \overline{M}'\,;\,\mathsf{let}\,X = p.\mathsf{call\_}\overline{a}\_c\_m(o,\vec{v})\,;\vec{C}')}\ \overline{a} \neq \emptyset$$

(W-DYN-MSG2)
$$\frac{\overline{D} \Vdash \mathsf{advice}(c::m) = [\overline{a}\,;\,\_] \qquad \overline{D} \Vdash \mathsf{genCallMth}(c::m;\,\overline{a}) = \overline{M} \qquad \overline{D} \Vdash \mathsf{wstack}(p\{\,\vec{C}\,\}) = (\overline{M}'\,;\,\vec{C}')}{\overline{D} \Vdash \mathsf{wstack}(p\{\,\mathsf{let}\,X = o : c.m(\vec{v})\,;\vec{C}\,\}) = (\overline{M}, \overline{M}'\,;\,\mathsf{skip}\,;\mathsf{let}\,X = p.\mathsf{call\_}\overline{a}\_c\_m(o,\vec{v})\,;\vec{C}')}\ \overline{a} \neq \emptyset$$

(W-STC-MSG)
$$\frac{\overline{D} \Vdash \mathsf{wstack}(p\{\,\vec{C}\,\}) = (\overline{M}\,;\,\vec{C}')}{\overline{D} \Vdash \mathsf{wstack}(p\{\,\mathsf{let}\,X = o.c :: m(\vec{v})\,;\vec{C}\,\}) = (\overline{M}\,;\,\mathsf{skip}\,;\mathsf{let}\,X = o.c :: m(\vec{v})\,;\vec{C}')}$$

(W-OTHER)
$$\frac{\text{no other command rule applies} \qquad \overline{D} \Vdash \mathsf{wstack}(p\{\,\vec{C}\,\}) = (\overline{M}\,;\,\vec{C}')}{\overline{D} \Vdash \mathsf{wstack}(p\{\,B\,\vec{C}\,\}) = (\overline{M}\,;\,B\,\vec{C}')} \qquad \frac{}{\overline{D} \Vdash \mathsf{wstack}(p\{\}) = (\emptyset\,;\,\emptyset)}\ \text{(W-NONE)}$$

(GEN-EXEC1)
$$\frac{\overline{D} \ni \_\,\mathsf{advice}\,b(\vec{X}) : r\ \mathsf{at}\ \_\,\{\,\vec{C}\,\} \qquad \nexists b' \in \overline{b'}.\ \overline{D} \ni \mathsf{replace\ advice}\,b'(\_) : \_\ \mathsf{at}\ \_\,\{\,\_\,\} \qquad \overline{D} \Vdash \mathsf{wstack}(\mathsf{this}\{\,\vec{C}[{}^{\mathsf{this}}\!/_{\mathsf{target}}]\,\}) = (\overline{M}\,;\,\vec{C}')}{\overline{D} \Vdash \mathsf{genExecMth}(c::m;\,\overline{b}) = \overline{M}, \mathsf{protected\ Object\ method\ exec\_}\overline{b}(\vec{X}) : r\,\{\,\vec{C}'\,\}}\ \begin{array}{l}\overline{b} = \underline{b}, \overline{b'} \\ b \prec \overline{b'}\end{array}$$

(GEN-EXEC2)
$$\frac{\overline{D} \ni \_\,\mathsf{advice}\,b(\vec{X}) : r\ \mathsf{at}\ \_\,\{\,\vec{C}\,\} \qquad \exists b' \in \overline{b'}.\ \overline{D} \ni \mathsf{replace\ advice}\,b'(\_) : \_\ \mathsf{at}\ \_\,\{\,\_\,\} \qquad \overline{D} \Vdash \mathsf{wstack}(\mathsf{this}\{\,\vec{C}[{}^{\mathsf{this}}\!/_{\mathsf{target}}, {}^{\mathsf{this}:c.m[\emptyset\,;\,\overline{b'}]}\!/_{\mathsf{proceed}}]\,\}) = (\overline{M}\,;\,\vec{C}') \qquad \overline{D} \Vdash \mathsf{genExecMth}(c::m;\,\overline{b'}) = \overline{M}'}{\overline{D} \Vdash \mathsf{genExecMth}(c::m;\,\overline{b}) = \overline{M}, \overline{M}', \mathsf{protected\ Object\ method\ exec\_}\overline{b}(\vec{X}) : r\,\{\,\vec{C}'\,\}}\ \begin{array}{l}\overline{b} = \underline{b}, \overline{b'} \\ b \prec \overline{b'}\end{array}$$

(GEN-CALL1)
$$\frac{\bar{D} \ni \_ \text{ advice } a(\vec{X}) : r \text{ at } \_ \{ \vec{C} \} \quad \bar{D} \Vdash \text{wstack}(\text{this}\{ \vec{C}[^{y}/_{\text{target}}, {}^{y.m}/_{\text{proceed}}] \}) = (\bar{M}' ; \vec{C}')}{\bar{D} \Vdash \text{genCallMth}(c :: m ; a) = \bar{M}', \text{ protected Object method call\_}a\text{\_}c\text{\_}m(y : c, \vec{X}) : r \{ \vec{C}' \}}$$

(GEN-CALL2)
$$\frac{\bar{D} \ni \_ \text{ advice } a(\vec{X}) : r \text{ at } \_ \{ \vec{C} \} \quad \bar{D} \Vdash \text{advice}(c :: m) = [\_ ; b] \quad \bar{D} \Vdash \text{wstack}(\text{this}\{ \vec{C}[^{y}/_{\text{target}}, {}^{\text{this}:c.m[\vec{a}' ; \bar{b}]}/_{\text{proceed}}] \}) = (\bar{M}' ; \vec{C}') \quad \begin{array}{l} \bar{a} = a, \vec{a}' \\ a \prec \vec{a}' \\ \vec{a}' \neq \emptyset \end{array}}{\bar{D} \Vdash \text{genCallMth}(c :: m ; \bar{a}) = \bar{M}', \text{ protected Object method call\_}\bar{a}\text{\_}c\text{\_}m(y : c, \vec{X}) : r \{ \vec{C}' \}}$$

THEOREM (5; WEAVING PRESERVES TYPABILTY).
For any statically weavable P, if $\Vdash P$: Prog *and* $\text{weave}(P) = P'$ *then* $\vdash P'$: Prog.

Proof. We divide method names into *generated names* of the form $\text{call\_}\bar{a}\text{\_}c\text{\_}m$ and $\text{exec\_}\bar{b}$, and *user names*. We assume without loss of generality, that P only contains user names.

Let $P = (\bar{D} \vdash \bar{H})$ and $P' = (\bar{D}' \vdash \bar{H}')$, and (without loss of generality, from Lemma 14) assume D is coherent. Then we have that:

$$\bar{D}; \bar{Z} \Vdash \bar{D} : \text{Dec} \qquad \bar{D}; \bar{Z} \Vdash \bar{H} : \text{Heap} \qquad \text{objects}(\bar{H}) = \bar{Z}$$
$$\bar{D} \Vdash \text{wdec}(\bar{D}) = \bar{D}' \qquad \bar{D} \Vdash \text{wheap}(\bar{H}) = \bar{H}' \qquad \text{objects}(\bar{H}') = \bar{Z}$$

and we show the following properties:

1. $\bar{D} \Vdash c$: Class if and only if $\bar{D}' \vdash c$: Class.
   Direct.
2. $\bar{D} \vdash t <: s$ if and only if $\bar{D}' \vdash t <: s$.
   Follows from property 1.
3. $\bar{D} \Vdash f$: Fld t for s in c if and only if $\bar{D}' \vdash f$: Fld t for s in c.
   Follows from property 2.
4. For any user method m, $\bar{D} \Vdash m$: Mth$(\vec{u}) : r$ for s in c
   if and only if $\bar{D}' \vdash m$: Mth$(\vec{u}) : r$ for s in c.
   Follows from property 2.
5. For any generated method m, if either:
   (a) $\bar{D} \Vdash \text{wstack}(p\{ \vec{B} \}) = (\bar{M} ; \vec{C})$, or
   (b) $\bar{D} \Vdash \text{genCallMth}(c :: m ; \bar{a}) = \bar{M}$, or
   (c) $\bar{D} \Vdash \text{genExecMth}(c :: m ; \bar{b}) = \bar{M}$
   and either:
   (a) $\bar{D} \Vdash \text{wstack}(p'\{ \vec{B}' \}) = (\bar{M}' ; \vec{C}')$, or
   (b) $\bar{D} \Vdash \text{genCallMth}(c' :: m' ; \bar{a}') = \bar{M}'$, or
   (c) $\bar{D} \Vdash \text{genExecMth}(c' :: m' ; \bar{b}') = \bar{M}'$
   and $M_0 \in \bar{M}$ and $M_0 = \text{protected } \_ \text{ method } m(\_) : \_ [\_ ; \_]$ and $M_0' \in \bar{M}'$
   and $M_0' = \text{protected } \_ \text{ method } m(\_) : \_ [\_ ; \_]$ then $M_0 = M_0'$.
   An induction on the weaving algorithm, in the case of (W-GEN-EXEC2) using the fact that (W-ADV-MSG1) removes c and m from $o : c.m[\emptyset ; \bar{b}]$.

6. For any generated method m, if $\overline{D} \Vdash \mathrm{wmth}(c;\, M) = \overline{M}$ and $M_0 \in \overline{M}$ and $M_0 = \mathsf{protected}\ \_\ \mathsf{method}\ m(\_) : \_\ [\_\,;\,\_]$ and $\overline{D} \Vdash \mathrm{wmth}(c';\, M') = \overline{M}'$ and $M_0' \in \overline{M}'$ and $M_0' = \mathsf{protected}\ \_\ \mathsf{method}\ m(\_) : \_\ [\_\,;\,\_]$ then $M_0 = M_0'$.
   Follows from property 5.
7. If $\overline{D};\, \overline{Z} \Vdash v\ :\ t$ then $\overline{D}';\, \overline{Z} \vdash v\ :\ t$.
   Follows from property 2.
8. If $\overline{D} \Vdash \mathrm{genCallMth}(c :: m;\, \overline{a}) = \overline{M}$ and $\overline{D} \Vdash \mathrm{advice}(c :: m) = [\overline{a}'\,;\,\_]$ and $\overline{a}' = \_, \overline{a}$ and $\overline{D} \Vdash m\ :\ \mathsf{Mth}(\vec{u}) : r\ \mathsf{for}\ s\ \mathsf{in}\ c$
   then $\vec{M} \ni \mathsf{protected\ Object\ method\ call}\_\overline{a}\_c\_m(y : c, \vec{x} : \vec{u}) : r\ \{\vec{C}\}$.
   Direct, making use of Lemma 13.
9. If $\overline{D} \Vdash \mathrm{genExecMth}(c :: m;\, \overline{b}') = \overline{M}$ and $\overline{D} \Vdash \mathrm{advice}(c :: m) = [\_\,;\,\overline{b}']$ and $\overline{b}' = \_, \overline{b}$ and $\overline{D} \Vdash m\ :\ \mathsf{Mth}(\vec{u}) : r\ \mathsf{for}\ s\ \mathsf{in}\ c$ and $\exists b \in \overline{b}.\ \overline{D} \ni \mathsf{replace\ advice}\ b(\_) : \_\ \mathsf{at}\ \_\ \{\_\}$
   then $\vec{M} \ni \mathsf{protected\ Object\ method\ exec}\_b(\vec{x} : \vec{u}) : r\ \{\vec{C}\}$.
   Direct, making use of Lemma 13.
10. If $\overline{D} \Vdash \mathrm{advice}(c :: m) = [\_\,;\,\overline{b}']$ and $\overline{b}' = \_, \overline{b}$ and $\overline{D} \Vdash m\ :\ \mathsf{Mth}(\vec{u}) : r\ \mathsf{for}\ s\ \mathsf{in}\ c$ and $\exists b \in \overline{b}.\ \overline{D} \ni \mathsf{replace\ advice}\ b(\_) : \_\ \mathsf{at}\ \_\ \{\_\}$
    then $\overline{D}' \vdash \mathsf{exec}\_b\ :\ \mathsf{Mth}(\vec{u}) : r\ \mathsf{for}\ s\ \mathsf{in}\ c$.
    Follows from property 9.
11. If $\overline{D};\, \overline{Z} \Vdash S\ :\ \mathsf{Stk\ replace}() : r\ \mathsf{for}\ d$ and $\overline{D};\, \overline{Z} \Vdash p\ :\ d$ and $\overline{D}' \ni \mathsf{class}\ d <: \_\ \{\_\ \overline{M}\}$ and $\overline{D} \Vdash \mathrm{wstack}(p\{S\}) = (\overline{M}\,;\, S')$ then $\overline{D};\, \overline{Z} \vdash S\ :\ \mathsf{Stk}\ r\ \mathsf{for}\ d$.
    An induction on S, making use of properties 8, 10, 18 and 19.
12. If $\overline{D} \Vdash \mathrm{genCallMth}(c :: m;\, \overline{a}) = \overline{M}$ and $\overline{D}' \ni \mathsf{class}\ c <: \_\ \{\_\ \overline{M}\}$ and $\overline{D} \Vdash m\ :\ \mathsf{Mth}(\vec{u}) : r\ \mathsf{for}\ s\ \mathsf{in}\ c$ and $\forall a \in \overline{a}.\ \overline{D} \Vdash c :: m \in \mathrm{calladv}(a)$
    then $\overline{D}';\, \overline{Z} \vdash \overline{M}\ :\ \mathsf{Mth\ in}\ c$.
    Uses Lemma 15, together with properties 11 and 14.
13. If $\overline{D} \Vdash \mathrm{genExecMth}(c :: m;\, \overline{b}) = \overline{M}$ and $\overline{D}' \ni \mathsf{class}\ c <: \_\ \{\_\ \overline{M}\}$ and $\overline{D} \Vdash m\ :\ \mathsf{Mth}(\vec{u}) : r\ \mathsf{for}\ s\ \mathsf{in}\ c$ and $\forall b \in \overline{b}.\ \overline{D} \Vdash c :: m \in \mathrm{execadv}(b)$ and $\exists b \in \overline{b}.\ \overline{D} \ni \mathsf{replace\ advice}\ b(\_) : \_\ \mathsf{at}\ \_\ \{\_\}$ then $\overline{D}';\, \overline{Z} \vdash \overline{M}\ :\ \mathsf{Mth\ in}\ c$.
    Uses Lemma 15, together with properties 11 and 14.
14. If $\overline{D};\, \overline{Z} \Vdash S\ :\ \mathsf{Stk\ replace}() : r\ \mathsf{for}\ d$ and $\overline{D};\, \overline{Z} \Vdash p\ :\ d$ and $\overline{D}' \ni \mathsf{class}\ d <: \_\ \{\_\ \overline{M}\}$ and $\overline{D} \Vdash \mathrm{wstack}(p\{S\}) = (\overline{M}\,;\, S')$ then $\overline{D};\, \overline{Z} \vdash \overline{M}\ :\ \mathsf{Mth\ in}\ .$
    An induction on S, making use of property 12.
15. If $\overline{D} \Vdash F\ :\ \mathsf{Fld}$ then $\overline{D}' \vdash F\ :\ \mathsf{Fld}$.
    Follows from property 1.
16. If $\overline{D} \Vdash \mathrm{wmth}(c;\, M) = \overline{M}$ and $\overline{D}' \ni \mathsf{class}\ c <: \_\ \{\_\ \overline{M}\}$ and $\overline{D};\, \overline{Z} \Vdash M\ :\ \mathsf{Mth\ in}\ c$ then $\overline{D}';\, \overline{Z} \vdash \overline{M}\ :\ \mathsf{Mth\ in}\ c$.
    Follows from property 13 and Lemma 5.
17. If $\overline{D} \vdash c$ is sane then $\overline{D}' \vdash c$ is sane.
    Follows from properties 2, 3, 4 and 6.
18. $\overline{D}';\, \overline{Z} \vdash \overline{D}'\ :\ \mathsf{Dec}$.
    Follows from properties 15, 16 and 17.
19. $\overline{D}';\, \overline{Z} \vdash \overline{H}'\ :\ \mathsf{Heap}$.
    Follows for object declarations from properties 1, 7 and 3; follows for thread declarations from property 11.

The result follows from properties 18 and 19.