Typed Parametric Polymorphism for Aspects

Radha Jagadeesan [a,1] Alan Jeffrey [b,1] James Riely [a,2]

[a] *School of CTI, DePaul University, Chicago, Illinois*

[b] *Security Technology Research, Bell labs, Lucent Technologies*

Abstract

We study the incorporation of generic types in aspect languages. Since advice acts like method update, such a study has to accommodate the subtleties of the interaction of classes, polymorphism and aspects. Indeed, simple examples demonstrate that current aspect compiling techniques do not avoid runtime type errors.

We explore type systems with polymorphism for two models of parametric polymorphism: the type erasure semantics of Generic Java, and the type carrying semantics of designs such as generic C#. Our main contribution is the design and exploration of a source-level type system for a parametric OO language with aspects. We prove progress and preservation properties.

We believe our work is the first source-level typing scheme for an aspect-based extension of a parametric object-oriented language.

Key words: Aspect-oriented programming, Typing, Generic types.

1 Introduction

Aspects have emerged as a powerful tool in the design and development of systems [1–6]. A profiling example from the AspectJ tutorials illustrates the use of aspects and helps to introduce the basic vocabulary. Suppose class *L* realizes a useful library, and we want to obtain timing information about a method `foo()` of *L*. With aspects this can be done by writing *advice* specifying that, whenever `foo` is called, the current time should be logged, `foo` should be executed, and then the current time should again be logged. It is indicative of the power of aspects that (a) the profiling code is localized in the advice, and (b) the responsibility for profiling all `foo()` calls resides with the compiler and/or runtime environment. The latter

[1] Research supported in part by NSF CyberTrust 0430175

[2] Research supported in part by NSF CAREER 0347542

ensures that the developer of the library need not worry about advice that may be written in the future. In [7] this notion is called *obliviousness*. However, in writing the logging advice, one must identify the pieces of code, using *pointcuts*, that need to be logged. In [7] this notion is called *quantification*.

Aspects provide general and paradigm-independent mechanisms for representing and composing crosscutting concerns such as logging. Aspect-oriented extensions have been developed for object-oriented [2,8,9], imperative [10,11], and functional languages [12,13]. There is also emerging research into the use of aspects at the requirement and the architecture level (e.g. see the proceedings of the workshop series on early aspects).

The diversity of applications testify to the success of the aspect approach. See [14] for an early systematic survey; we mention only a few examples here. Aspects address inheritance anomalies (see [15] for a survey) of concurrent object-oriented programming [16]. They provide a basic ingredient for variability management in programming with features [17,18]. Aspects enable useful refactoring of (operating) systems [11] and middleware [19] code. They also support program visualization by enabling program monitoring [20].

Much recent research in aspect programming languages aims to further facilitate change through increased language expressiveness by enhancing the quantification mechanism. To name but a few, there are explorations of virtual machine support for dynamic (i.e. incorporated at runtime) join points [21], the treatment of pointcuts as functional queries [22], description of pointcuts that can operate on distributed code [23], domain-specific and user-defined extensions to the pointcut language [24], and logic-based metaprogramming mechanisms [25].

The above research focuses primarily on facilitating the expression and composition of aspects in programming languages. Instead, we are interested in exploring aspect language mechanisms that are both dynamic and *safe*, so as to not compromise the trustworthiness of the system. Our main new technical development is a source-level type system for aspect languages that incorporates the parametric features of object-oriented languages. We show that type safety is preserved by reduction.

One motivation is the use of aspect languages in security applications (e.g., see [26]). Consider the use of Inlined Reference Monitors (IRMs, see [27] for a survey) to enforce fine-grained, application-specific access policies. Aspects enable elegant implementations of IRMs: the IRM writer writes the security policy as an aspect, and the aspect weaver merges the checking code into the application itself to produce a secured application. This application requires that basic safety guarantees are provided by the aspect language.

More generally, we are interested in aspect language mechanisms that support a principle identified in [28]: *services may be refined as long as the original promises*

are still upheld. In this paper, we focus on the simple invariants of memory safety (programs can only access appropriate memory locations) and control safety (programs can only transfer control to appropriate program points).

The study of expressive type systems for aspect languages was recognized as an important research problem early on [29]. Aspectual collaborations [30] provide compelling evidence supporting the utility of generic advice. PolyAML [31] explores polymorphic types in a functional language with explicit programmer annotations of the control points at which advice may be added. Polymorphism has also been studied in the implementation of the functional language Aspectual Caml [32], where aspects also interact with type inference and curried functions.

However, we believe our work is the first source-level typing scheme for an aspect-based extension of a parametric object-oriented language. We explore type systems with polymorphism for two models of parametric polymorphism: the type erasure semantics of Generic Java [33] and Pizza [34], and the type carrying semantics of designs such as generic C# [35]. Our formal investigations provide another data point in the ongoing argument between the two styles of parametric polymorphism.

Such a study has to accommodate the subtleties of the interaction of classes, polymorphism and aspects. Advice, at a first approximation, acts like method update [36]. Hence, it needs to be treated carefully from a typing point of view. Simple examples demonstrate that current aspect compiling techniques do not avoid runtime type errors.

Our contribution is timely, as full source-level support for the generic features of Java 1.5 is just now available in AspectJ. In particular, Java 1.5's addition of covariant return types, and the type erasure implementation of generics, presents problems for typesafe aspect-oriented languages, which we discuss in Section 2.

We conclude this introduction by addressing the impact of our study on the basic obliviousness and quantification criteria on aspect languages. From a programming point of view, type systems support an abstract view of the interface and provide a way to enforce the principle of Least Privilege [37]: Throughout execution, each principal should be accorded the minimum access necessary to accomplish its task. Type systems are thus an example of a programming feature that "derives power precisely from what they prevent some other programmer from doing" [38]. Superficially, these restrictions contradict uninhibited and unrestricted obliviousness and quantification, leading to an impression that the very idea of AOP is incompatible with typing and cannot coexist with it.

We disagree with this conclusion. We take the point of view that in practice, the tension is mitigated by the increasing expressiveness of type systems. Such a viewpoint is not new [30]: in object-oriented programming dynamic dispatch yields obliviousness [7], and modular reasoning requires the taming of this obliviousness by expressive behavioral types (to ensure subtypes do not violate the contract of su-

pertypes) [39]. Currently, advances in type systems are making it easier to specify and implement type systems, leading to wider acceptance of rich programmer annotations: the parametric types of Java and C# bear witness to this. We believe that the finer control and stronger guarantees provided by rich type systems compensate for the restrictions imposed by typing on AOP mechanisms.

Organization of the paper. In the next section, we use simple examples to illustrate the issues. To make this paper self contained, Section 3 describes Featherweight Generic Java (FGJ) [40], which is the basis for our aspect language. The following section presents the dynamic semantics of Aspect FGJ (AFGJ). In Section 5 we present a typing system which is sound for the type carrying semantics. Finally in Section 6 we describe a more restrictive type system which is sound for the type erasure semantics.

2 Examples

We study the static semantics of an aspect language. We provide type systems that satisfy two fundamental properties:

- well typed programs do not get stuck in type errors, and
- well typing is preserved by reduction.

These properties show that our language is typesafe [41]. Our key technical contribution is the identification of *sufficient* typing restrictions on pointcuts and advice to establish the above properties for both (a) the type preserving semantics of C# and (b) the type erasure semantics of Java.

Our aspect language is an extension of cast-free Featherweight Generic Java (FGJ) [40]. Our focus is on a direct source-level semantics of a Java-like language; we do not build on work on translations of class-based languages into polymorphic λ-calculi or object-based languages [42–46]. While FGJ is similar in spirit to Classic Java [47], Java$_s$ [48] and Middleweight Java [49], our choice of FGJ as a basis for our formal study is based on the extant analysis of parametric types in FGJ. A similar study has been conducted of generics in the .NET common language runtime [50].

The focus of this paper is on the key ideas underlying the aspect extension. our pointcut language is simple: it captures method execution and includes vararg parameters and boolean operators. We have not considered call pointcuts or temporal operators such as AspectJ's `cflow`; we do not address inner classes [51], wildcards [52], or type inference for generic methods.

Before our formal study, we sketch language features and design issues of an aspect language with parametricity. We present examples in the familiar style of AspectJ; in Section 4, we describe how these can be translated into AFGJ. The goals of these examples are to demonstrate that:

- generic advice is useful, even in the presence of nongeneric base classes,
- advice complicates covariant return types, and
- advice complicates the type erasure semantics used by Generic Java.

In the remainder of this paper, we formalize the design of a generic aspect language in light of the issues discussed here.

Generic aspects are useful even for nongeneric base classes. Suppose that we want to transform the result of all parameterless methods in a class `C` by passing them as parameters to a method `D.do_after`. Parameterizing on the return type, we might write:

```
class C {
  String  f() { return "1"; }
  Integer g() { return  1;  }
}
aspect CodingAfter<T> {
   T around(): execution(T C.*())  {
      return new D().do_after(proceed());
} }
```

The current stable AspectJ compiler (build 20060217002528) forces generic aspects to be abstract, and thus will not compile the preceding code.

Without a generic typing scheme one must either duplicate the aspect code for each return type in `C` or one must type the around advice at `Object`, thus losing static type guarantees. Indeed, the following code compiles successfully using the current stable AspectJ compiler, but yields a `ClassCastException` when executing `new C().f()`:

```
aspect WrongReturnType {
  Object around(): execution(* C.*()) { return 2; }
}
```

This difficulty arises with argument types as well as return types. We demonstrate this using a standard crosscutting concern: synchronization. (The example incorporates non-functional features which are not expressible in pure AFGJ.) Consider a class `Out` with signature:

```
void writeByte(Byte x)
void writeInteger(Integer x)
void writeCharacter(Character x)
...
```

Suppose we would like to write a locking aspect, which adds synchronization to each method in `Out`:

```
aspect LockingStream<T> {
  void around(T x): args(x) && execution(void Out.*(*)) {
      synchronized(this) { proceed(x); }
} }
```

Without generics, the advice would have to be written

```
aspect NonGenericLockingStream {
  void around(Object x): args(x) && execution(void Out.*(*)) {
      synchronized(this) { proceed(x); }
} }
```

but it is difficult to see how to allow `NonGenericLockingStream` without also allowing:

```
aspect WrongParameterType {
  void around(Object x): args(x) && execution(void Out.*(*)) {
      proceed(new Object { "hello" });
} }
```

This advice would cause a call to `writeByte` to proceed with an argument containing a `String` object, and hence generate a run-time type error.

These examples illustrate that generic advice can be valuable even with nongeneric base classes.

Aspects complicate covariant return types. The difficulties are even more apparent in the context of *covariant return types*. In Java 1.5, a subclass may refine the return type of a method it overrides, as in `CarFactory.build()`:

```
class Vehicle { ... }
class Car extends Vehicle { ... }
class VehicleFactory { Vehicle build() { ... } }
class CarFactory extends VehicleFactory { Car build() { ... } }
```

Unfortunately, such covariant return types conflict with aspects:

```
class Motorcycle extends Vehicle { ... }
aspect AlwaysBuildMotorcyle {
  Vehicle around(): execution(* VehicleFactory.build()) {
    return new Motorcycle();
} }
```

Considering `AlwaysBuildMotorcycle` and the classes it mentions in isolation, one might believe that the aspect is typesafe (and indeed it is for Java 1.4). How-

ever, AlwaysBuildMotorcyle applies not only to VehicleFactory, but also to all subclasses, causing CarFactory to produce Motorcycles rather than Cars.

We propose using generics as a solution to the general problem of interaction between aspects and return types. Parametricity ensures type safety, banning examples such as AlwaysBuildMotorcycle while still allowing aspects to apply to methods with different return types, as in CodingAfter.

Expressiveness considerations. We argue informally that these typing constraints are not unduly restrictive. [3] Typical logging, monitoring and checks of safety conditions before executing the method (c.f. the IRM application discussed in the introduction) are parametric in the return type and are unaffected. On the other hand, general advice that replaces the method with potentially unrelated new behavior is severely restricted by our typing schemes.

Contravariant argument types. In contrast to Java and C# methods, C# delegates also permit contravariant argument types. Our formal treatment in this paper does not address this feature. In the conclusions, we outline the issues raised by the interaction of aspects with contravariant argument types.

Aspects complicate type erasure. The type erasure semantics of a generic languages such as Java replaces any generic type by an upper bound, for example

```
class List<T extends Comparable<T>> { T[] contents; ... }
```

is type erased by replacing T by its upper bound Comparable to become:

```
class List { Comparable[] contents; ... }
```

This type erasure semantics is used by Java [40], but not by C# [50]. As a result, Java does not have full type information available at runtime, whereas C# does.

Consider a parameterized list class List<T> with a method max that returns a new object whose encapsulated array has each of its indices set to be the greater of contents[i] and the argument x[i]. Such a method requires an ordering on T, which is ensured using the type bound Comparable<T> [40]:

```
class List<T extends Comparable<T>> {
  T[] contents; ...
  List<T> max(List<T> x) {
    // general code for general types
} }
```

[3] We revisit this issue, using the vocabulary of [53], in the technical development of Sections 5 and 6.

In the case of a boolean list, bit operations might be used to obtain a more efficient implementation. The following skeleton captures such an aspect:

```
aspect BooleanMax {
  List<Boolean> around(List<Boolean> x): args(x) &&
    execution(List<Boolean> List<Boolean>.max(List<Boolean>)) {
    // special code for boolean arguments
} }
```

It is important to see that whether the pointcut in the advice above fires or not depends on the type of the argument to `max`. For example, consider the following generic program:

```
List<Integer> a  = new List<Integer>();
List<Integer> am = a.max(a);
List<Boolean> b  = new List<Boolean>();
List<Boolean> bm = b.max(b);
```

The call to `b.max(b)` will cause the specialized advice in `BooleanMax` to be called, but the call to `a.max(a)` will be unadvised. However, this program is type erased to become:

```
List a  = new List();
List am = a.max(a);
List b  = new List();
List bm = b.max(b);
```

After type erasure, it is impossible to distinguish the two method calls. Consequently for such aspects, type information must be present at run time. A generic aspect language for C# may allow such aspects (since C# does not use type erasure), but a generic aspect language for Java must not.

In our generic aspect language for type erasure semantics, we restrict pointcuts as follows: all parameters to classes (such as List) in a pointcut must be variables. However, this restriction by itself is not sufficient to ensure type safety.

Generic aspects complicate type erasure. Consider a pair class:

```
class Pair<T,U> {
  T first; U second; ...
  T getFirst() { return first; }
}
```

It is reasonable to allow generic advice on such generic objects:

```
class Log<T> { T log(T x) { ... } }
aspect Logging<T,U> {
  T around(): execution(T Pair<T,U>.first()) {
```

```
        return new Log().log(proceed());
    } }
```

Even in the presence of type erasure, such generic advice is typesafe. However, it is possible to write generic advice which is not safe in the presence of type erasure. One source of nonsafety is nonlinear uses of type variables, as in the pointcut `execution(T Pair<T,T>.getFirst())` This pointcut matches `Pair<Integer, Integer>` but not to `Pair<Integer, String>`, thus exposing dependence on instance information.

3 Featherweight Generic Java

We present the syntax, dynamic and static semantics of cast-free Featherweight Generic Java (FGJ) [40], which forms the basis for the aspect language presented in later sections. FGJ restricts Java 1.5 to its bare essentials. For example, FGJ has no mutable state, no interfaces, no overloading and a restricted form of constructors. Similar restrictions are also present in the other analysis of generics, such as [50]. To make this paper self contained, we include all necessary definitions, referring the reader to the original paper [40] for a full and detailed exposition. [4]

FGJ disallows `instanceof` and reflection; we have chosen additionally to eliminate casting, for a number of reasons:

- Casting is required in full Featherweight Generic Java because the main result in [40] is the soundness of the translation into Featherweight Java, which introduces casts. We do not discuss the translation in this paper, and so we do not require casts.
- The type rules for casting in Featherweight Generic Java are complex, due to the requirement that type erasure be sound. This complexity is reflected in the proofs of subject reduction, and would obscure the central point of this paper: the interaction of generics and aspects.
- The concerns raised by casting are orthogonal to aspects; we expect that, suitably adapted to account for the complexities mentioned above, our results apply to the language with casting.

[4] Our presentation differs from that of [40] in several superficial respects. We have chosen different metavariable names. We use Java type declaration conventions throughout; for example, method types are written "$R(\bar{P})$" rather than "$\bar{P} \to R$" and term variable bindings are written "$T\ x$" rather than "$x : T$". We have unified method body lookup and method type lookup into a single definition. We explicitly mention the global set of declarations $\mathscr{D}$ (called CT in [40]) when it is used in definitions. We define evaluation contexts explicitly, with a single evaluation rule, rather than listing a separate evaluation rule for each form of context. Our typing rules explicitly require that all variables in environments and parameter lists be distinct; eg, we disallow `class` $c\langle X \lhd C, X \lhd D\rangle \cdots$.

Let c and d range over class names (including the reserved name `Object`). Let f, g and h range over field names. Let ℓ range over method names. Let x range over term variables (including the reserved variables `this` and `target`). And let X, Y, Z and W range over type variables.

The syntax of the language is given below. For any syntactic category with typical element e, we write $\bar{e}$ for an ordered sequence $e_1, e_2, \ldots, e_n$ with n implicit; the element separator may be a space, comma, or semicolon, depending on context. We use i, ranging between 1 and n, to pick out a particular element e_i. We occasionally extend this convention across binary constructs; for example, we write $\bar{X} \lhd \bar{C}$ for $X_1 \lhd C_1, \ldots, X_n \lhd C_n$, and $\bar{T}\ \bar{x}$ for $T_1\ x_1, \ldots, T_n\ x_n$. We write "$\bullet$", or simply a blank space, for the empty sequence.

FGJ SYNTAX

$C, D, E ::= c\langle \bar{T} \rangle$	*Nonvariable Types*
$P, Q, R, S, T, U, V ::= X \mid C$	*Types*
$O ::= \texttt{new}\ C(\bar{O})$	*Values*
$M, N, L ::=$	*Terms*
$\quad x$	*Term Variable*
$\quad M.f$	*Field Access*
$\quad M.\ell\langle \bar{T} \rangle(\bar{N})$	*Method Call*
$\quad \texttt{new}\ C(\bar{N})$	*Object*
$\mathscr{E} ::=$	*Evaluation Contexts*
$\quad [].f$	*Field Access*
$\quad [].\ell\langle \bar{T} \rangle(\bar{N})$	*Method Call Target*
$\quad M.\ell\langle \bar{T} \rangle(\bar{N}, [], \bar{N}')$	*Method Call Argument*
$\quad \texttt{new}\ C(\bar{N}, [], \bar{N}')$	*Object*
$\mathscr{D} ::= \texttt{class}\ c\langle \bar{X} \lhd \bar{C} \rangle \lhd D\{\bar{T}\ \bar{f};\ \kappa\ \bar{\mu}\}$	*Top Level Declarations*
$\kappa ::= c(\bar{T}\ \bar{f})\{\texttt{super}(\bar{g});\ \texttt{this}.\bar{h}=\bar{h};\}$	*Constructor Declarations*
$\mu ::= \langle \bar{X} \lhd \bar{C} \rangle R\ \ell(\bar{P}\ \bar{x})\{M\}$	*Method Declarations*
$\Delta ::= \bullet \mid \Delta, X \lhd C$	*Type Environment*
$\Gamma ::= \bullet \mid \Gamma, T\ x$	*Term Environment*

As in [40], we write "$\lhd$" for "`extends`". In a method declaration "$\langle \bar{X} \lhd \bar{C} \rangle R\ \ell(\bar{P}\ \bar{x})\{$ `return`M`;`$\}$" we elide the `return` and semicolon. We also drop angled brackets when there are no type parameters; for example, we write `Object` rather than `Object`$\langle\rangle$.

The variables $\bar{X}$ are bound in the class declaration $\texttt{class}\ c\langle \bar{X} \lhd \bar{C} \rangle \lhd D\{\bar{T}\ \bar{f};\ \kappa\ \bar{\mu}\}$; the scope is $\bar{C}$, D, $\bar{T}$, κ and $\bar{\mu}$. The variables $\bar{X}$, $\bar{x}$ and the reserved variable `this` are bound in the method declaration $\langle \bar{X} \lhd \bar{C} \rangle R\ \ell(\bar{P}\ \bar{x})\{M\}$; the scope of $\bar{X}$ is $\bar{C}$, R, $\bar{P}$ and

M; the scope of $\bar{x}$ and `this` is M. We identify syntax up to renaming of bound variables. For any syntactic category with typical element e, we write $\mathsf{fv}(e)$ for the set of free variables occurring in e. Substitution of terms for term variables and types for type variables is as usual. We write substitutions postfix; for example, we write $M[{}^{T}/_{X}, {}^{N}/_{x}]$ for the term derived from M by simultaneously replacing occurrences of X with T and occurrences of x with N. We treat environments as mappings, writing $\Delta(X)$ for the bound of X in Δ, and writing $\Gamma(x)$ for the type of x in Γ.

To clarify definitions, we elide irrelevant elements of the syntax. For example, consider a method declared $\langle \bar{X} \triangleleft \bar{C} \rangle R\ \ell(\bar{P}\ \bar{x})\{M\}$. If we are interested only in the body of the method, we may write the declaration simply as $\langle \bar{X} \rangle\ \ell(\bar{x})\{M\}$. If we are interested only in the type, we may write the same declaration as $\langle \bar{X} \triangleleft \bar{C} \rangle R\ \ell(\bar{P}) \cdots$.

ASSUMPTION 3.1 (FIXED DECLARATIONS). Evaluation and other relations are defined with respect to a fixed set of declarations. To avoid repeating the set of declarations, we fix a set $\mathscr{D}$ of declarations for the remainder of the paper. As in [40], we assume that `Object` is not declared and that the induced subclass relation is antisymmetric. [5] □

FGJ Dynamics. The evaluation relation is defined using auxiliary definitions for field and method lookup. These definitions are also used in typing; thus they are parameterized by a typing environment Δ, which is empty during evaluation.

LOOKUP $(\Delta \vdash \mathsf{fields}(T) = \bar{T}\ \bar{f})$ $(\Delta \vdash \mathsf{meth}(T.\ell) = \langle \bar{Y} \triangleleft \bar{E} \rangle R(\bar{P}\ \bar{x})\{M\})$

(FIELD-OBJECT)

$$\frac{}{\Delta \vdash \mathsf{fields}(\texttt{Object}) = \bullet}$$

(METHOD-THIS)

$$\frac{\mathscr{D} \ni \texttt{class}\ c\langle \bar{X} \rangle \cdots \{ \cdots \langle \bar{Y} \triangleleft \bar{E} \rangle R\ \ell(\bar{P}\ \bar{x})\{M\} \}}{\Delta \vdash \mathsf{meth}(c\langle \bar{V} \rangle.\ell) = (\langle \bar{Y} \triangleleft \bar{E} \rangle R(\bar{P}\ \bar{x})\{M\})[{}^{\bar{V}}/_{\bar{X}}]}$$

(FIELD-THIS-SUPER)

$$\frac{\mathscr{D} \ni \texttt{class}\ c\langle \bar{X} \rangle \triangleleft D\{\bar{T}\ \bar{f}; \cdots\} \quad \vdash \mathsf{fields}(D[{}^{\bar{V}}/_{\bar{X}}]) = \bar{S}\ \bar{g}}{\Delta \vdash \mathsf{fields}(c\langle \bar{V} \rangle) = \bar{S}\ \bar{g}, \bar{T}[{}^{\bar{V}}/_{\bar{X}}]\ \bar{f}}$$

(METHOD-SUPER)

$$\frac{\mathscr{D} \ni \texttt{class}\ c\langle \bar{X} \rangle \triangleleft D\{\cdots \bar{\mu}\} \quad \ell \text{ not defined in } \bar{\mu} \quad \vdash \mathsf{meth}(D[{}^{\bar{V}}/_{\bar{X}}].\ell) = \langle \bar{Y} \triangleleft \bar{E} \rangle R(\bar{P}\ \bar{x})\{M\}}{\Delta \vdash \mathsf{meth}(c\langle \bar{V} \rangle.\ell) = \langle \bar{Y} \triangleleft \bar{E} \rangle R(\bar{P}\ \bar{x})\{M\}}$$

(FIELD-VAR)

$$\frac{\vdash \mathsf{fields}(\Delta(X)) = \bar{T}\ \bar{f}}{\Delta \vdash \mathsf{fields}(X) = \bar{T}\ \bar{f}}$$

(METHOD-VAR)

$$\frac{\vdash \mathsf{meth}(\Delta(X).\ell) = \langle \bar{Y} \triangleleft \bar{E} \rangle R(\bar{P}\ \bar{x})\{M\}}{\Delta \vdash \mathsf{meth}(X.\ell) = \langle \bar{Y} \triangleleft \bar{E} \rangle R(\bar{P}\ \bar{x})\{M\}}$$

Field lookup collects the fields of a class with those of its superclasses. Method lookup finds the most specialized class that declares a method. We now define evaluation.

[5] The subclass relation is the smallest preorder on class names induced by the rule: $c \leq d$ if $\mathscr{D} \ni \texttt{class}\ c\langle \cdots \rangle \triangleleft d \cdots$. These restrictions on $\mathscr{D}$ are required for weak confluence and progress; they ensure that field and method lookup are deterministic and total for well typed terms.

FGJ EVALUATION $(M \to M')$

$$\frac{\mathsf{fields}(C) = \bar{f}}{\texttt{new}\ C(\bar{N}).f_i \to N_i}\ \text{(EVAL-FIELD)} \qquad \frac{M = \texttt{new}\ C(\cdots) \quad \mathsf{meth}(C.\ell) = \langle \bar{X} \rangle (\bar{x})\{L\}}{M.\ell\langle \bar{V} \rangle [\,](\bar{N}) \to L[{}^{\bar{V}}/_{\bar{X}}, {}^{M}/_{\texttt{this}}, {}^{\bar{N}}/_{\bar{x}}]}\ \text{(EVAL-METHOD)} \qquad \frac{M \to M'}{\mathscr{E}[M] \to \mathscr{E}[M']}\ \text{(EVAL-CONTEXT)}$$

Note that in EVAL-METHOD the residual term is subject to three substitutions: for the type parameters $\bar{X}$, and for the term parameters $\bar{x}$ and `this`. The definition of contexts allows nondeterministic evaluation, since all method and constructor parameters are treated equally. As usual, we write $\mathscr{E}[M]$ for $\mathscr{E}[{}^{M}/_{[\,]}]$.

FGJ Statics. Typing uses several auxiliary definitions, which characterize subtyping, well formed types and environments, and well formed overriding. [6]

AUXILIARY JUDGMENTS $(\Delta \vdash T <: T')\ \ (\Delta \vdash T)\ \ (\Delta \vdash \mathsf{ok})\ \ (\Delta; \Gamma \vdash \mathsf{ok})$

$$\frac{}{\Delta \vdash X <: \Delta(X)}\ \text{(SUB-VAR)} \qquad \frac{\mathscr{D} \ni \texttt{class}\ c\langle \bar{X} \rangle \lhd D \cdots}{\Delta \vdash c\langle \bar{V} \rangle <: D[{}^{\bar{V}}/_{\bar{X}}]}\ \text{(SUB-CLASS)} \qquad \frac{}{\Delta \vdash T <: T}\ \text{(SUB-REFLEX)} \qquad \frac{\Delta \vdash T <: T' \quad \Delta \vdash T' <: T''}{\Delta \vdash T <: T''}\ \text{(SUB-TRANS)}$$

$$\frac{X \in \mathsf{dom}(\Delta)}{\Delta \vdash X}\ \text{(TYPE-VAR)} \qquad \frac{\mathscr{D} \ni \texttt{class}\ c\langle \bar{X} \lhd \bar{C} \rangle \cdots \quad \Delta \vdash \bar{V} \quad \Delta \vdash \bar{V} <: \bar{C}[{}^{\bar{V}}/_{\bar{X}}]}{\Delta \vdash c\langle \bar{V} \rangle}\ \text{(TYPE-CLASS)} \qquad \frac{}{\Delta \vdash \texttt{Object}}\ \text{(TYPE-OBJECT)}$$

$$\frac{\forall i\,.\, X_1 \lhd C_1, \ldots, X_n \lhd C_n \vdash C_i \quad \forall i,j\,.\, X_i = X_j \text{ implies } i = j}{X_1 \lhd C_1, \ldots, X_n \lhd C_n \vdash \mathsf{ok}}\ \text{(ENV-TYPE)} \qquad \frac{\Delta \vdash \mathsf{ok}}{\Delta; \bullet \vdash \mathsf{ok}}\ \text{(ENV-EMPTY)} \qquad \frac{\Delta; \Gamma \vdash \mathsf{ok} \quad \Delta; \Gamma \vdash T}{\Delta; \Gamma, T\ x \vdash \mathsf{ok}}\ x \notin \mathsf{dom}(\Gamma)\ \text{(ENV-TERM-VAR)}$$

Subtyping is induced from variable and class declarations. Well formed types include declared variables, `Object`, and parameterized classes which satisfy the required constraints. An environment is well formed if all variables are unique and if all of the types it contains are well formed.

WELL FORMED OVERRIDING $(\vdash \langle \bar{Y} \lhd \bar{E} \rangle R(\bar{P})\ \mathsf{can\,override}\ D.\ell)$

$$\frac{\mathsf{meth}(D.\ell)\ \text{undefined}}{\vdash \langle \bar{Y} \lhd \bar{E} \rangle R(\bar{P})\ \mathsf{can\,override}\ D.\ell}\ \text{(OVERRIDE-UNDEFINED)} \qquad \frac{\mathsf{meth}(D.\ell) = \langle \bar{Y} \lhd \bar{E} \rangle R(\bar{P}) \cdots \quad \bar{Y} \lhd \bar{E} \vdash R' <: R}{\vdash \langle \bar{Y} \lhd \bar{E} \rangle R'(\bar{P})\ \mathsf{can\,override}\ D.\ell}\ \text{(OVERRIDE-DEFINED)}$$

[6] We group related judgments; for example, we write "$\Delta \vdash \bar{V}$" to abbreviate $\forall i\,.\,\Delta \vdash V_i$; we also write $\Delta; \Gamma \vdash (M, \bar{N}) : (T, \bar{P})$ to abbreviate $\Delta; \Gamma \vdash M : T$ and $\Delta; \Gamma \vdash \bar{N} : \bar{P}$.

Overriding is covariant in return type, but invariant in generic type bounds and parameter types. The typing rules for class and method declarations are given next.

TOP LEVEL DECLARATION, METHOD TYPING $(\vdash \mathscr{D})$ $(\vdash \mu : \mathsf{ok\,in}\, c\langle \bar{X} \triangleleft \bar{C} \rangle \triangleleft D)$

(DEC-CLASS)

$$\frac{\begin{array}{l}\bar{X} \triangleleft \bar{C} \vdash \bar{C}, D, \bar{T} \\ \mathsf{fields}(D) = \bar{S}\ \bar{g} \\ \bar{X} \triangleleft \bar{C};\ \bar{S}\ \bar{g}, \bar{T}\ \bar{f} \vdash \mathsf{ok} \\ \vdash \bar{\mu} : \mathsf{ok\,in}\, c\langle \bar{X} \triangleleft \bar{C} \rangle \triangleleft D \\ \kappa = c(\bar{S}\ \bar{g}, \bar{T}\ \bar{f})\{\texttt{super}(\bar{g})\texttt{;this.}\bar{f}\texttt{=}\bar{f}\texttt{;}\}\end{array}}{\vdash \texttt{class}\ c\langle \bar{X} \triangleleft \bar{C} \rangle \triangleleft D\{\bar{T}\ \bar{f};\ \kappa\ \bar{\mu}\}}$$

(DEC-METHOD)

$$\frac{\begin{array}{l}\bar{X} \triangleleft \bar{C}, \bar{Y} \triangleleft \bar{E} \vdash \bar{E}, \bar{P}, R \\ \bar{X} \triangleleft \bar{C}, \bar{Y} \triangleleft \bar{E};\ \bar{P}\ \bar{x}, T\ \texttt{this} \vdash M : R' \\ \bar{X} \triangleleft \bar{C} \vdash R' <: R \\ \vdash \langle \bar{Y} \triangleleft \bar{E} \rangle R(\bar{P})\ \mathsf{can\,override}\ D.\ell\end{array}}{\vdash \langle \bar{Y} \triangleleft \bar{E} \rangle R\,\ell(\bar{P}\ \bar{x})\{M\} : \mathsf{ok\,in}\, c\langle \bar{X} \triangleleft \bar{C} \rangle \triangleleft D}$$

The third premise of DEC-CLASS ensures that all type variables and fields are unique by requiring that the corresponding environment be well formed — without this restriction on fields, EVAL-FIELD is potentially nondeterministic. A similar restriction is imposed by the second premise of DEC-METHOD, since well formed terms must have well formed environments (Lemma B.4).

ASSUMPTION 3.2 (DECLARATIONS ARE TYPED). We require that each declarations in the global declaration environment $\mathscr{D}$ be well formed. That is, in all definitions and results, we assume that $\vdash \mathscr{D}_i$, for each $\mathscr{D}_i$ in $\mathscr{D}$. □

The rules for terms are as follows.

TERM TYPING $(\Delta; \Gamma \vdash M : T)$

(TERM-VAR)

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash \mathsf{ok} \\ \Gamma(x) = T\end{array}}{\Delta; \Gamma \vdash x : T}$$

(TERM-FIELD)

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash M : T \\ \Delta \vdash \mathsf{fields}(T) = \bar{S}\ \bar{f}\end{array}}{\Delta; \Gamma \vdash M.f_i : S_i}$$

(TERM-METHOD)

$$\frac{\begin{array}{l}\Delta \vdash \bar{V} \\ \Delta \vdash \mathsf{meth}(T.\ell) = \langle \bar{Y} \triangleleft \bar{E} \rangle R(\bar{P}) \cdots \\ \Delta; \Gamma \vdash (M, \bar{N}) : (T, \bar{P}') \\ \Delta \vdash (\bar{V}, \bar{P}') <: (\bar{E}, \bar{P})[\bar{V}/\bar{Y}]\end{array}}{\Delta; \Gamma \vdash M.\ell\langle \bar{V} \rangle(\bar{N}) : R}$$

(TERM-OBJECT)

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash \mathsf{ok} \quad \Delta \vdash C \\ \vdash \mathsf{fields}(C) = \bar{S}\ \bar{f} \\ \Delta; \Gamma \vdash \bar{N} : \bar{S}' \\ \Delta \vdash \bar{S}' <: \bar{S}\end{array}}{\Delta; \Gamma \vdash \texttt{new}\ C(\bar{N}) : C}$$

The rules TERM-VAR and TERM-OBJECT ensure that only well formed environments can be used in typing a term.

As shown in [40], the language enjoys type preservation and progress properties. In addition, evaluation is confluent. Proof sketches are provided in Appendix B. For full proofs, see [40]. The statement of progress uses the notion of value O, defined in FGJ Syntax.

THEOREM 3.3 (PRESERVATION). *If* $\vdash M : S$ *and* $M \to N$ *then* $\exists T.\ \vdash T <: S$ *and* $\vdash N : T$.

THEOREM 3.4 (PROGRESS). *If* $\vdash M$ *then either* M *is a value or* $\exists N.\ M \to N$.

THEOREM 3.5 (WEAK CONFLUENCE). *If $M \to N_1$ and $M \to N_2$ then $\exists L \,.\, N_1 \to^* L$ and $N_2 \to^* L$.*

4 Aspect FGJ

The syntax of Aspect FGJ is adapted from that of AspectJ. Advice supports type parameters similar to those of a polymorphic method. Pointcuts are unnamed; they must be specified directly in advice declarations. We restrict attention to around advice and execution pointcuts. We also remove the redundant binders in pointcuts.

For example the AspectJ term

```
aspect a<X extends V> {
  R around(T t, P x): target(t) && args(x) && execution(R T.*(..)) {
    return proceed(t,x);
} }
```

is rendered as "$\texttt{advice a}\langle \texttt{X} \triangleleft \texttt{V} \rangle \texttt{ R(P x) : exe R T.*(*)}\{\texttt{proceed(x)}\}$".

AFGJ extends FGJ with forms for advice declaration and for proceeding to the next declared advice. We describe the operational semantics as a small step semantics; thus, we need the syntax to describe running code. To this end, we add a form for *advised calls* (from [54]), which are terms in the process of executing advice. Advised calls contain a list of *advice applications* which name the advice that remains to be run. The initial list of advice applications for a given method call is determined by the pointcuts contained in aspect declarations.

ASPECT FGJ SYNTAX

$A, B ::= a\langle \bar{T} \rangle$	*Advice Application*
$M, N, L ::= \cdots$	*Terms*
$\quad M.\ell\langle \bar{T} \rangle\,[\bar{A}]\,(\bar{N})$	*Advised Call*
$\quad \texttt{proceed}(\bar{N})$	*Proceed Call*
$\mathscr{E} ::= \cdots$	*Evaluation Contexts*
$\quad [].\ell\langle \bar{T} \rangle\,[\bar{A}]\,(\bar{N})$	*Advised Call Target*
$\quad M.\ell\langle \bar{T} \rangle\,[\bar{A}]\,(\bar{N}, [], \bar{N'})$	*Advised Call Argument*
$\quad \texttt{proceed}(\bar{N}, [], \bar{N'})$	*Proceed Call Argument*
$\mathscr{D} ::= \cdots$	*Top Level Declarations*
$\quad \texttt{advice}\ a\langle \bar{X} \triangleleft \bar{C} \rangle\, R\,(\bar{P}\ \bar{x}) : \phi\{M\}$	*Advice Declaration*
$\phi, \psi, \rho ::=$	*Pointcuts*
$\quad \texttt{exe}\ R\ T.\ell\langle \bar{V} \rangle\,(\bar{P})$	*Method Execution*
$\quad \texttt{exe}\ R\ T.\ell\langle \bar{V} \rangle\,(\bar{P}, *)$	*Vararg Method Execution*
$\quad \texttt{exe}\ R\ T.*\,(\bar{P})$	*Wildcard Execution*

$\texttt{exe}\, R\, T.*(\bar{P},*)$	*Vararg Wildcard Execution*
ϕ && ψ	*And*
ϕ || ψ	*Or*
`false`	*False*
`true`	*True*
$\Gamma ::= \cdots$	*Term Environment*
$\Gamma, R\,\texttt{proceed}(\bar{P})$	*Proceed Declaration*

AFGJ requires three reserved variables (`this`, `target` and `proceed`) in addition to the reserved name `Object`. The variables $\bar{X}$, $\bar{x}$ and the reserved variables `target` and `proceed` are bound in the declaration $\texttt{advice}\, a\langle \bar{X} \triangleleft \bar{V} \rangle\, R(\bar{P}\,\bar{x}) : \phi\,\{M\}$; the scope of $\bar{X}$ is $\bar{V}$, R, $\bar{P}$, ϕ and M; the scope of $\bar{x}$, `target` and `proceed` is M.

Proceed declarations record the type of a method's parameters and return value; they are used only in typing, discussed in the next section.

Pointcuts. The events which can trigger advice are method calls. In our language, pointcuts are terms in a positive boolean logic (i.e. no negation) with atoms describing method calls. To indicate that a method call satisfies a pointcut, we use a *pointcut logic*.

POINTCUT LOGIC $(\phi \vDash \psi)$

$\texttt{exe}\, R\, T.\ell\langle \bar{V} \rangle(\bar{P},\bar{Q}) \vDash \texttt{exe}\, R\, T.\ell\langle \bar{V} \rangle(\bar{P},*)$	$\rho \vDash \texttt{true}$	$\texttt{false} \vDash \rho$
$\texttt{exe}\, R\, T.\ell\langle \bar{V} \rangle(\bar{P},\bar{Q},*) \vDash \texttt{exe}\, R\, T.\ell\langle \bar{V} \rangle(\bar{P},*)$	$\rho \vDash \phi$ && ψ	if $\rho \vDash \phi$ and $\rho \vDash \psi$
$\texttt{exe}\, R\, T.\ell\langle \bar{V} \rangle(\bar{P}) \vDash \texttt{exe}\, R\, T.*(\bar{P})$	$\rho \vDash \phi$ || ψ	if $\rho \vDash \phi$
$\texttt{exe}\, R\, T.\ell\langle \bar{V} \rangle(\bar{P},\bar{Q}) \vDash \texttt{exe}\, R\, T.*(\bar{P},*)$	$\rho \vDash \phi$ || ψ	if $\rho \vDash \psi$
$\texttt{exe}\, R\, T.\ell\langle \bar{V} \rangle(\bar{P},\bar{Q},*) \vDash \texttt{exe}\, R\, T.*(\bar{P},*)$	ϕ && $\psi \vDash \rho$	if $\phi \vDash \rho$
$\texttt{exe}\, R\, T.*(\bar{P},\bar{Q}) \vDash \texttt{exe}\, R\, T.*(\bar{P},*)$	ϕ && $\psi \vDash \rho$	if $\psi \vDash \rho$
$\texttt{exe}\, R\, T.*(\bar{P},\bar{Q},*) \vDash \texttt{exe}\, R\, T.*(\bar{P},*)$	ϕ || $\psi \vDash \rho$	if $\phi \vDash \rho$ and $\psi \vDash \rho$

Note that, in the above rules, $\bar{Q}$ may be empty. The wildcard "$*$" in the arguments permits a form of varargs in pointcuts.

Most of the use of the logic is with fully concrete method calls on the left, even though the logic itself is presented more generally. Pointcut ϕ is satisfied by the method call $T.\ell\langle \bar{V} \rangle$ if $\texttt{exe}\, R\, T.\ell\langle \bar{V} \rangle(\bar{P}) \vDash \phi$, where $\bar{P}$ and R are the parameter and return types declared for the method. T is the dynamic type of the receiver, so we effectively model dynamic dispatch to choose the applicable advice — this matches the semantics of execution pointcuts in AspectJ like languages. In particular, we can't statically know what advice applies at a given call site.

Since we are modeling single dispatch languages, however, the parameter types $\bar{P}$ (and the return type R) are determined by the declared type of a method $\ell\langle \bar{V} \rangle$ in a

class $c\langle\bar{T}\rangle$, rather than the actual parameter types.

All the pointcuts used in Section 2 are expressible in the above logic. We illustrate the pointcut logic by a series of examples to explicate the design issues.

EXAMPLE 4.1. The pointcut `exe Object` c`.*(*)` captures all messages sent to instances of c that are declared with return type `Object`; the receiving object's actual class must be exactly c. The first wildcard captures all method names. The wildcard in the argument position illustrates the use of the varargs facility in pointcuts, and enables the capture of all methods without worrying about the number and type of parameters. □

The pointcut logic forces all types to match exactly. It is intuitively clear that without negation, it is not possible to define exact matching from a construct that matches a type and all its subtypes. But what about the converse? We now show that genericity compensates for the demands placed by the requirement of exact matching by allows us to express downward type closure quite easily.

EXAMPLE 4.2. The generic pointcut $\langle X\triangleleft c, Y\triangleleft$`Object`$\rangle$`exe` Y X`.*(*)` captures all messages sent to instances of *any subclass* of c, regardless of the declared return type. Type variables in a pointcut such as this are expressed in the advice declaration, as in

$$\texttt{advice}\ a\langle X\triangleleft c, Y\triangleleft\texttt{Object}\rangle\ Y\ \texttt{():exe}\ Y\ X\texttt{.*(*)}\cdots.$$

The pointcut $\langle X\triangleleft$`Object`$, Y\triangleleft$`Object`$\rangle$`exe` Y X`.*(*)` captures all methods in all subclasses of `Object`. □

Negation in the pointcut logic. We disallow negation in pointcuts since it interferes with the typing systems given in later sections: see section 5. We now discuss the limitations on expressivity that are caused by this design.

Our logic directly captures one of the primary uses of negation, which is to match a type exactly rather than including subtypes. More generally, using disjunction, this enables us to write pointcuts that pick out any finite subset of types.

The true limitations of the absence of negation would be seen were we to include interfaces. In this case, the absence of negation would prevent us from writing pointcuts that isolate some subsets of types, e.g. all types that do *not* implement an interface.

Dynamics. If parameterized advice $a\langle\bar{X}\rangle$ fires, then the pointcut must also generate a binding for each X_i — there is nothing else to provide constraints. This leads to the definition of *advice lookup*, given in the next table. This definition is also used in typing; thus it is parameterized on a typing environment, Δ, which is empty during evaluation. In this paper, we do not address the algorithmic issues related to determining the type parameters of advice.

ADVICE LOOKUP $(\Delta \vdash T.\ell\langle\bar{V}\rangle \text{ advised by } a\langle\bar{U}\rangle)$

$$\frac{\begin{array}{l}\mathscr{D} \ni \texttt{advice}\ a\langle\bar{X} \triangleleft \bar{C}\rangle : \phi\ \cdots \\ \Delta \vdash \bar{U} <: \bar{C}[\bar{U}/\bar{X}] \quad \Delta \vdash \bar{U} \\ \Delta \vdash \mathsf{meth}(T.\ell) = \langle\bar{Y}\rangle R(\bar{P}) \cdots \\ \texttt{exe}\ R[\bar{V}/\bar{Y}]\ T.\ell\langle\bar{V}\rangle(\bar{P}[\bar{V}/\bar{Y}]) \vDash \phi[\bar{U}/\bar{X}]\end{array}}{\Delta \vdash T.\ell\langle\bar{V}\rangle \text{ advised by } a\langle\bar{U}\rangle}$$

Note that the constraints on type variables in advice may not be unique. For example consider advice a, with type variables $\langle X, Y\rangle$ and pointcut $\phi = \texttt{exe}\ R\ X.\ell\langle Y\rangle() \ \texttt{||}\ \texttt{exe}\ R\ Y.\ell\langle X\rangle()$. The event $\texttt{exe}\ R\ c.\ell\langle d\rangle()$ is advised by both $a\langle c,d\rangle$ and $a\langle d,c\rangle$. As a result, advice lookup is nondeterministic. Determinism is recovered via typing, described in the next section.

AFGJ EVALUATION (EVAL-FIELD AND EVAL-CONTEXT FROM FGJ)

(EVAL-LOOKUP)

$$\frac{[\bar{A}] = \left[a\langle\bar{U}\rangle \;\middle|\; \begin{array}{l}\mathscr{D} \ni \texttt{advice}\ a\ \cdots \\ \vdash C.\ell\langle\bar{V}\rangle \text{ advised by } a\langle\bar{U}\rangle\end{array}\right]}{M.\ell\langle\bar{V}\rangle(\bar{N}) \rightarrow M.\ell\langle\bar{V}\rangle\,[\bar{A}]\,(\bar{N})} \quad M = \texttt{new}\ C(\cdots)$$

(EVAL-ADVICE)

$$\frac{\mathscr{D} \ni \texttt{advice}\ a\langle\bar{X}\rangle\,(\bar{x})\cdots\{L\}}{M.\ell\langle\bar{V}\rangle\,[a\langle\bar{U}\rangle,\bar{A}]\,(\bar{N},\bar{N}') \rightarrow L[\bar{U}/\bar{X}, M/_{\texttt{target}}, \bar{N}/_{\bar{x}}, {}^{M.\ell\langle\bar{V}\rangle\,[\bar{A}]\,(\bar{N}')}/_{\texttt{proceed}}]}$$

(EVAL-METHOD)

$$\frac{\mathsf{meth}(C.\ell) = \langle\bar{X}\rangle\,(\bar{x})\,\{L\}}{M.\ell\langle\bar{V}\rangle\,[]\,(\bar{N}) \rightarrow L[\bar{V}/\bar{X}, M/_{\texttt{this}}, \bar{N}/_{\bar{x}}]} \quad M = \texttt{new}\ C(\cdots)$$

The evaluation strategy is adapted from our previous work [54]. EVAL-LOOKUP uses comprehension syntax [55] to denote the sequence of advice declared in $\mathscr{D}$ that advises the method call, in declaration order. This rule uses the dynamic type of the object as determined by the constructor. As discussed earlier, since this is execution advice, following languages such as AspectJ, the dynamic type of the receiver is used to fetch the matching advice.[7] EVAL-ADVICE then executes the advice, in order, passing the remaining advice through to `proceed`; this is accomplished using the special substitution form defined below. Finally, EVAL-METHOD executes the method body once the advice list is empty.

PROCEED SUBSTITUTION $(L[{}^{M.\ell\langle\bar{V}\rangle\,[\bar{A}]\,(\bar{N}')}/_{\texttt{proceed}}] = L')$

The substitution is homomorphic for all term constructs but $\texttt{proceed}(\bar{N})$.

[7] We consider only execution pointcuts in this paper. For AspectJ's *call* pointcuts, the *static* type of the receiver is used to fetch matching advice [54].

$$\texttt{proceed}(\bar{N})\left[^{M.\ell\langle\bar{V}\rangle\,[\bar{A}]\,(\bar{N}')}/_{\texttt{proceed}}\right] = M.\ell\langle\bar{V}\rangle\,[\bar{A}]\,(\bar{N},\bar{N}')$$

EVAL-ADVICE and the proceed substitution, in combination, pass all the arguments to the proceed variable. The splitting of the argument list in EVAL-ADVICE accommodates the flavor of varargs in our formalism, and is illustrated in the following examples.

EXAMPLE 4.3. Recall from example 4.2 that the form `X extends C` can be used to define a pointcut that matches all subtypes of `C`.

```
class C {
   int const()   { return 42; }
   int id(int x) { return x; }
}
class D extends C { ... }
advice <X extends C> int a(): int X.*(*) {
   return proceed() + 1;
}
```

Evaluation of the term `new D().const()` proceeds as follows:

```
   new D().const()
-> new D().const[a<D>]()          [eval-lookup]
-> new D().const[]() + 1          [eval-advice + proceed substitution]
-> 42 + 1                         [eval-method]
```

Evaluation of the term `new D().id(5)` proceeds as follows:

```
   new D().id(5)
-> new D().id[a<D>](5)            [eval-lookup]
-> new D().id[](5) + 1            [eval-advice + proceed substitution]
-> 5 + 1                          [eval-method]
```

These two evaluations illustrate how the same advice — in this case `a` — influences methods with different numbers of parameters — in this case `const()` and `id(int)` — exploiting the varargs in our formalism. □

EXAMPLE 4.4. `before` and `after` advice as classically construed in AspectJ rely on side effects, and thus are not very useful in the pure calculus that we consider. We consider a pure functional form of `after` advice which transforms the result of method call. The general form is $\texttt{after}\ \langle\bar{X}\triangleleft\bar{V}\rangle R\ a(\bar{P}\,\bar{x})\,[\phi]\,\{M\}$, where the special variable `result` is allowed to occur in M with type R. This can be encoded as $\texttt{advice}\ a\langle\bar{X}\triangleleft\bar{V}\rangle\,R(\bar{P}\,\bar{x}) : \phi\{M[^{\texttt{proceed}(\bar{x})}/_{\texttt{result}}]\}$. The form of varargs supported by our system, as illustrated by the earlier discussion of EVAL-ADVICE, is crucial to ensuring that the *same* definition of "after" advice can be used for *any* method whose list of parameters has $\bar{P}\,\bar{x}$ as a prefix.

Similarly, the general form $\texttt{before}\ \langle \bar{X} \triangleleft \bar{V} \rangle R\ a(\bar{P}\ \bar{x})\ [\phi]\ \{\bar{M}\}$, where each M_i represents a transformation of argument x_i, can be encoded as:

$$\texttt{advice}\ a\langle \bar{X} \triangleleft \bar{V} \rangle\ R(\bar{P}\ \bar{x}) : \phi\ \{\texttt{proceed}(\bar{M})\}$$ □

5 Aspect FGJ Statics: Type Carrying Semantics

In this section, we discuss the typing of Featherweight Aspect GJ for the case when precise type information is carried at runtime.

To guarantee unique bindings for advice variables, we must ensure that each variable is used in every satisfiable pointcut. To do this, we formalize the notion of disjunctively-closed free variables first. The base cases of the following definition for atoms yields the free type variables. In a conjunction, the type variables can be bound in either conjunct. In a disjunction, the type variables have to be bound in *both* disjuncts. Thus, we are requiring (roughly) that all type variables in a pointcut occur in every disjunctive subterm.

The following definition relies on the absence of negation in the pointcut logic.

DISJUNCTIVELY-CLOSED FREE VARIABLES $(\mathsf{dcfv}(\phi) = \bar{X})$

$$\begin{aligned}
\mathsf{dcfv}(\texttt{exe}\ R\ T.*(\bar{P},*)) &= \mathsf{fv}(R) \cup \mathsf{fv}(T) \cup \mathsf{fv}(\bar{P}) \\
\mathsf{dcfv}(\texttt{exe}\ R\ T.*(\bar{P})) &= \mathsf{fv}(R) \cup \mathsf{fv}(T) \cup \mathsf{fv}(\bar{P}) \\
\mathsf{dcfv}(\texttt{exe}\ R\ T.\ell\langle \bar{V} \rangle(\bar{P},*)) &= \mathsf{fv}(R) \cup \mathsf{fv}(T) \cup \mathsf{fv}(\bar{V}) \cup \mathsf{fv}(\bar{P}) \\
\mathsf{dcfv}(\texttt{exe}\ R\ T.\ell\langle \bar{V} \rangle(\bar{P})) &= \mathsf{fv}(R) \cup \mathsf{fv}(T) \cup \mathsf{fv}(\bar{V}) \cup \mathsf{fv}(\bar{P}) \\
\mathsf{dcfv}(\phi\ \texttt{||}\ \psi) &= \mathsf{dcfv}(\phi) \cap \mathsf{dcfv}(\psi) \qquad \mathsf{dcfv}(\texttt{false}) = \emptyset \\
\mathsf{dcfv}(\phi\ \texttt{\&\&}\ \psi) &= \mathsf{dcfv}(\phi) \cup \mathsf{dcfv}(\psi) \qquad \mathsf{dcfv}(\texttt{true}) = \emptyset
\end{aligned}$$

The following proposition shows that this definition achieves its intention.

PROPOSITION 5.1 (DETERMINISTIC ADVICE LOOKUP).
Let $\mathscr{D} \ni \texttt{advice}\ a\langle \bar{Y} \triangleleft \bar{E} \rangle\ R(\bar{P}\ \bar{x}) : \phi\ \{M\}$. *If* $\mathsf{fv}(\phi) = \mathsf{dcfv}(\phi)$ *and* $\vdash T.\ell\langle \bar{V} \rangle$ advised by $a\langle \bar{U} \rangle$ *and* $\vdash T.\ell\langle \bar{V} \rangle$ advised by $a\langle \bar{U}' \rangle$ *then* $\bar{U} = \bar{U}'$.

The resulting type rules are described in the next table.

AFGJ TYPING (ALL RULES FROM FGJ TYPING)

(DEC-ADVICE)

$$\frac{\begin{array}{l}\phi \vDash \texttt{exe}\ R\ T\ .\ *\ (\bar{P}, *) \\ \mathsf{fv}(\phi) = \mathsf{dcfv}(\phi) = \{\bar{Y}\} \\ \bar{Y} \triangleleft \bar{E} \vdash T, \bar{E}, \bar{P}, R \\ \bar{Y} \triangleleft \bar{E};\ \bar{P}\ \bar{x}, T\ \texttt{target}, R\ \texttt{proceed}(\bar{P}) \vdash M : R' \\ \bar{Y} \triangleleft \bar{E} \vdash R' <: R\end{array}}{\vdash \texttt{advice}\ a\langle \bar{Y} \triangleleft \bar{E} \rangle\ R(\bar{P}\ \bar{x}) : \phi\{M\}}$$

(TERM-PROCEED)

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash \mathsf{ok} \\ \Gamma(\texttt{proceed}) = R(\bar{P}) \\ \Delta; \Gamma \vdash \bar{N} : \bar{P}' \\ \Delta \vdash \bar{P}' <: \bar{P}\end{array}}{\Delta; \Gamma \vdash \texttt{proceed}(\bar{N}) : R}$$

(TERM-ADVISED)

$$\frac{\begin{array}{l}\Delta \vdash C.\ell\langle \bar{V} \rangle\ \mathsf{advised\ by}\ \bar{A} \\ \Delta; \Gamma \vdash M.\ell\langle \bar{V} \rangle\ (\bar{N}) : R\end{array}}{\Delta; \Gamma \vdash M.\ell\langle \bar{V} \rangle\ [\bar{A}]\ (\bar{N}) : R}\ M = \texttt{new}\ C(\cdots)$$

(ENV-PROCEED)

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash \mathsf{ok} \quad \Delta; \Gamma \vdash \bar{P}, R \\ \texttt{proceed} \notin \mathsf{dom}(\Gamma)\end{array}}{\Delta; \Gamma, R\ \texttt{proceed}(\bar{P}) \vdash \mathsf{ok}}$$

TERM-PROCEED uses a well formed proceed declaration to type the result of a call to proceed. TERM-ADVISED checks the well typing of an advised method call using the well typing of the advice that are members in the list. In this rule we require that the term M be an object; this is not overly restrictive since advised method calls are intermediate results which occur only after the receiver's type is fully evaluated.

Most of the work is performed in the rule DEC-ADVICE. This rule requires that the advice satisfy the "Disjunctively-Closed Free Variables" condition discussed earlier. In addition, it addresses two further issues: constraints on pointcuts and constraints on return types.

Constraints on pointcuts. Pointcuts are not typed in DEC-ADVICE. Nonetheless, they are subject to two constraints.

- The requirement on disjunctively-closed free variables ensures that all the type variables in a pointcut are constrained by any method call triggering the advice.
- The requirement that $\phi \vDash \texttt{exe}\ R\ T\ .\ *\ (\bar{P}, *)$ ensures additionally that all triggers agree on types for `target` and for the parameters listed in the pointcut.

In particular, the second requirement is reflected in the definition of the pointcut logic that both $\phi \vDash \rho$ and $\psi \vDash \rho$ are required to satisfy $\phi \ ||\ \psi \vDash \rho$. This is illustrated by the following example

EXAMPLE 5.2. Recall that the pointcut

$$\phi = \langle X, Y \rangle (\texttt{exe}\ R\ X\ .\ \ell\langle Y \rangle\ ()\ ||\ \texttt{exe}\ R\ Y\ .\ \ell\langle X \rangle\ ())$$

is satisfied by the event $\texttt{exe}\ R\ c\ .\ \ell\langle d \rangle\ ()$ with two conflicting variable bindings: $\langle c, d \rangle$ and $\langle d, c \rangle$. Such a pointcut is disallowed by DEC-ADVICE since the two sides of the disjunction disagree on the type of the target. □

Constraints on the return type. DEC-ADVICE uses the type information recovered from the pointcut via $\phi \vDash \texttt{exe}\ R\ T\texttt{.*(}\bar{P}\texttt{,*)}$ to generate assumptions to type the body of the advice in $\bar{X}\triangleleft\bar{C}; \bar{P}\,\bar{x}, T\ \texttt{target}, R\ \texttt{proceed(}\bar{P}\texttt{)} \vdash M : S$. This impacts and interacts with (covariant) overriding, as illustrated by the following examples.

EXAMPLE 5.3. Consider the pointcut `exe Object c.foo(*)`. DEC-ADVICE in this case sets the return type, R, to `Object` and the rule permits the advice to return a result at any subtype of `Object`. Informally, this is typesafe since in this case the declared return type of the triggering method is exactly `Object`. □

EXAMPLE 5.4. The reasoning of the previous example extends to more general pointcuts. Consider the pointcut $\langle X\triangleleft c\rangle$`exe Object` X`.foo(*)`. Again in this case, DEC-ADVICE sets R to `Object` and permits the advice to return a result at any subtype of `Object`. Informally, even though the receiving object's actual class can be any subclass of c, this is also typesafe since the declared return type of the trigger method is (exactly) `Object`. □

EXAMPLE 5.5. Consider the generic pointcut $\langle X\triangleleft c, Y\triangleleft$`Object`$\rangle$`exe` $Y\ X$`.foo(*)`. In this case, DEC-ADVICE sets R to the type variable Y. DEC-ADVICE specifies typesafe ways for the use of the return values of the proceed in the advice body. The advice body can use the return value of proceed at the bound of the proceed variable, in this case `Object`. Furthermore, the advice body has to return a result at a subtype of Y, without knowing the exact type associated with Y. So, the advice body has to be parametric in the return value from proceed. This discussion is made concrete by the first example from Section 2, rendered in AFGJ as:

```
advice <R extends E> R CodingAfter: exe(R C.*()) {
      return new D().do_after(proceed());
}
```

In order for this advice to type, `D.do_after` must have type $\langle X\triangleleft E\rangle X$`()`. □

EXAMPLE 5.6. Consider the pointcut `exe(List<Boolean> List<Boolean>.max (List<Boolean>))`. In this case, DEC-ADVICE sets R to `List<Boolean>`, specifies that the advice body can use the return value of proceed at `List<Boolean>` and forces the advice body to return a result at a subtype of `List<Boolean>`. □

We revisit our informal arguments in the introduction — that our typing constraints are not unduly restrictive — using the vocabulary of [53] that classifies advice as *augmentation* advice (where the entire body of the method always executes), *narrowing* advice (either the entire body of the method executes or none of the body executes) and *replacement* advice (advice replaces the method with potentially unrelated new behavior). With the proviso that the advice body is generic in the return value from proceed, in the sense of example 5.5, augmentation and narrowing advice are permitted by our typing restrictions. Typical logging, monitoring and safety

checks clearly fall into this category. On the other hand, replacement advice on a method in a class is generally typesafe only in the restricted situation that the advice applies only to this class and not to any of its the subclasses.

Results. Theorem 3.3 (Preservation) and Theorem 3.4 (Progress) hold for AFGJ. See Appendix C for proofs. Unlike Theorem 3.5, the weak confluence result for AFGJ requires that terms be typed.

THEOREM 5.7 (AFGJ WEAK CONFLUENCE). *If* $\vdash M : T$*, for some* T*, and* $M \to N_1$ *and* $M \to N_2$ *then* $\exists L \,.\, N_1 \to^* L$ *and* $N_2 \to^* L$*.*

The proof of weak confluence relies on proposition 5.1. For details see Appendix C.

6 Aspect FGJ Statics: Type Erasure Semantics

The type system of the previous section guarantees deterministic advice lookup: that whenever advice $a\langle \bar{U} \rangle$ fires, the choice of each U_i is unique (proposition 5.1). Here we develop a type system that additionally guarantees that whether advice fires is independent of the type parameters — so the types do not need to be present at runtime and can be erased. Concretely, our typing rules ensure that if a piece of advice `advice` $a\langle \bar{X} \triangleleft \bar{C} \rangle$ `(...)` $: \phi \{ \ldots \}$ fires with types $\bar{V}$, the type information can be abstracted out (into $\bar{X}$) and replaced with any other types $\bar{U}$ which satisfy the constraints.

The erasure typing ($\Vdash$) includes all rules from the previous system ($\vdash$), but for DEC-ADVICE. The second premise of DEC-ADVICE now also includes pointcut typing.

ERASURE TYPING (ALL PRIOR RULES EXCEPT DEC-ADVICE)

(DEC-ADVICE)

$$\frac{\begin{array}{l}\phi \vDash \texttt{exe}\ R\ T\texttt{.*(}\bar{P}\texttt{,*)} \\ \bar{Y}\triangleleft\bar{E} \Vdash \phi \\ \bar{Y}\triangleleft\bar{E} \vdash T, \bar{E}, \bar{P}, R \\ \bar{Y}\triangleleft\bar{E}; \bar{P}\ \bar{x}, T\ \texttt{target}, R\ \texttt{proceed(}\bar{P}\texttt{)} \Vdash M : R' \\ \bar{Y}\triangleleft\bar{E} \vdash R' <: R\end{array}}{\Vdash \texttt{advice}\ a\langle\bar{Y}\triangleleft\bar{E}\rangle\ R(\bar{P}\ \bar{x}) : \phi\{M\}}$$

(PC-EXE-CLASS)

$$\frac{\begin{array}{l}\bar{X} \text{ and } \bar{Y} \text{ disjoint} \\ \mathscr{D} \ni \texttt{class}\ c\langle\bar{X}\triangleleft\bar{C}\rangle\cdots \\ \vdash \mathsf{meth}(c\langle\bar{X}\rangle.\ell) = \langle\bar{Y}\triangleleft\bar{E}\rangle R(\bar{P})\cdots \\ \bar{X}\triangleleft\bar{C}\ , \bar{Y}\triangleleft\bar{E}\ \vdash (\bar{C}, \bar{E}) <: (\bar{C}', \bar{E}')\end{array}}{\bar{X}\triangleleft\bar{C}', \bar{Y}\triangleleft\bar{E}' \Vdash \texttt{exe}\ R\ c\langle\bar{X}\rangle.\ell\langle\bar{Y}\rangle(\bar{P})}$$

(PC-EXE-VAR)

$$\frac{\begin{array}{l}W, Z, \bar{X}, \text{ and } \bar{Y} \text{ disjoint} \\ \mathscr{D} \ni \texttt{class}\ c\langle\bar{X}\triangleleft\bar{C}\rangle\cdots \\ \vdash \mathsf{meth}(c\langle\bar{X}\rangle.\ell) = \langle\bar{Y}\triangleleft\bar{E}\rangle R(\bar{P})\cdots \\ W\triangleleft R\ , Z\triangleleft c\langle\bar{X}\rangle, \bar{X}\triangleleft\bar{C}, \bar{Y}\triangleleft\bar{E}\ \vdash (\bar{E}, R) <: (\bar{E}', R')\end{array}}{W\triangleleft R', Z\triangleleft c\langle\bar{X}\rangle, \bar{X}\triangleleft\bar{C}, \bar{Y}\triangleleft\bar{E}' \Vdash \texttt{exe}\ W\ Z.\ell\langle\bar{Y}\rangle(\bar{P})}$$

(PC-WILD)

$$\frac{\begin{array}{l}W, Z, \bar{X}, \text{ and } \bar{Y} \text{ disjoint} \\ \mathscr{D} \ni \texttt{class}\ c\langle\bar{X}\triangleleft\bar{C}\rangle\cdots\end{array}}{W\triangleleft\texttt{Object}, Z\triangleleft c\langle\bar{X}\rangle, \bar{X}\triangleleft\bar{C}, \bar{Y}\triangleleft\texttt{Object} \Vdash \texttt{exe}\ W\ Z.\texttt{*(}\bar{Y}\texttt{)}}$$

(PC-FALSE) $\dfrac{}{\Vdash \texttt{false}}$ (PC-TRUE) $\dfrac{}{\Vdash \texttt{true}}$

(PC-AND)

$$\frac{\begin{array}{l}\bar{X} \text{ and } \bar{Y} \text{ disjoint} \\ \bar{X}\triangleleft\bar{C} \Vdash \phi \\ \bar{Y}\triangleleft\bar{D} \Vdash \psi\end{array}}{\bar{X}\triangleleft\bar{C}, \bar{Y}\triangleleft\bar{D} \Vdash \phi\ \texttt{\&\&}\ \psi}$$

(PC-OR)

$$\frac{\begin{array}{l}\Delta \Vdash \phi \\ \Delta \Vdash \psi\end{array}}{\Delta \Vdash \phi\ \texttt{||}\ \psi}$$

(PC-REORDER)

$$\frac{\begin{array}{l}\Delta\ \Vdash \phi \\ \Delta' \text{ is a permutation of } \Delta\end{array}}{\Delta' \Vdash \phi}$$

(PC-VARARGS-EXE)

$$\frac{\Delta \Vdash \texttt{exe}\ R\ T.\ell\langle\bar{E}\rangle(\bar{P}, \bar{Q})}{\Delta \Vdash \texttt{exe}\ R\ T.\ell\langle\bar{E}\rangle(\bar{P}\texttt{,*)}}$$

(PC-VARARGS-WILD)

$$\frac{\Delta \Vdash \texttt{exe}\ R\ T\texttt{.*(}\bar{P}, \bar{Q}\texttt{)}}{\Delta \Vdash \texttt{exe}\ R\ T\texttt{.*(}\bar{P}\texttt{,*)}}$$

The type system we develop for pointcuts has two rules for each atom in the pointcut logic. One rule applies when the target type is a class; the other rule applies when the target type is a variable.

The pointcut typing rules impose two kinds of restrictions to ensure that erasure of types does not affect the choice of triggered pointcuts. First, the parameters to classes are forced to be variables, since the instantiation of these type variables are not available at run time. Second, a linear discipline is enforced on type variables in pointcuts to eliminate hidden dependencies on type instantiations. This linear discipline is developed by analogy with intuitionist linear logic: the PC-AND rule corresponds to *tensor introduction* ($\otimes$), whereas the PC-OR rule corresponds to *with introduction* (&) and PC-REORDER corresponds to the multiset view of the context. Keeping with the linear discipline, weakening and strengthening are disallowed.

PC-EXE-CLASS forces the parameters to classes to be distinct variables via the restriction on valid class declarations. It also forces the type variables used in the result type of the method to be disjoint from the type parameters to the classes. PC-EXE-VAR imposes the additional restriction that the type variable used for the

class itself and the result type are also distinct variables. The differences between PC-WILD and PC-EXE-VAR are caused by the fact that the method is not specified in PC-WILD, obviating the need for those hypothesis in the type judgment.

The two PC-VARARGS rules permit vararg pointcuts to be typed if there is some instance of the varargs that permits the typing.

EXAMPLE 6.1. The pointcut `exe(List<Boolean> List<Boolean>.max(List <Boolean>))` is not typeable since PC-EXE-CLASS requires that all parameters to classes be variables. This example should be contrasted with example 5.6. □

EXAMPLE 6.2. The pointcut `exe(T Pair<T,T>.first())` is also untypeable. The second hypothesis in PC-EXE-CLASS requires `Pair<T,T>` to be part of a valid class declaration. A valid class declaration is not permitted to have repeated type variables in our system. PC-AND prevents us from using conjunction to achieve the same result, for example by using `exe(T Pair<T,U>.first()) && exe(U Pair<U,T>.first())`. □

EXAMPLE 6.3. On the affirmative side, the example encoding of `after` advice given previously — $\texttt{advice}\, a\langle \bar{X} \triangleleft \bar{C} \rangle\, R\,(\bar{P}\,\bar{x}) : \phi\,\{M[^{\texttt{proceed}(\bar{x})}/_{\texttt{result}}]\}$ — is typeable provided that the following provisos hold: M is typeable, $\bar{X}, \bar{P}$ are disjoint and $\bar{X} \triangleleft \bar{C} \Vdash \phi$. In the case where ϕ is of the form picking out a method call ℓ in class c, $\texttt{exe}\; R\; c\langle \bar{Z} \rangle . \ell \langle \bar{Y} \rangle (\bar{P})$, the last proviso implies that the type parameters to the class c are type variables $\bar{Z}$ and also that $\bar{Z}$ and $\bar{Y}$ disjoint. □

Results. The following lemma relates the erasure semantics to the type carrying semantics from Section 5.

PROPOSITION 6.4. *If* $\Delta; \Gamma \Vdash M : T$ *then* $\Delta; \Gamma \vdash M : T$.

Proof. The crucial lemma states that if $\bar{X} \triangleleft \bar{C} \Vdash \phi$ then $\mathsf{fv}(\phi) = \mathsf{dcfv}(\phi) = \{\bar{X}\}$. Given the change to DEC-ADVICE, this allows us to conclude that the erasure semantics ($\Vdash$) is more restrictive than the type carrying semantics ($\vdash$). □

Proposition 6.4 allows us to carry over the proofs of weak confluence, progress and preservation from the previous system. In the case of preservation, note that the changes in the type system only affect declarations, which do not evolve under evaluation.

We now argue that type erasure is safe for the typing system presented in this section by proving that runtime types do not affect evaluation. The following theorem is proved in Appendix D.

THEOREM 6.5 (PARAMETRICITY OF REDUCTION). *Suppose* $\bar{X} \triangleleft \bar{D} \Vdash M : T$ *and* $\vdash \bar{S}$ *and* $\vdash \bar{S} <: \bar{D}[\bar{V}/\bar{X}]$. *Then* $\vdash M[\bar{S}/\bar{X}] \to L$ *implies that* $L = N[\bar{S}/\bar{X}]$ *and for all* $\bar{S}'$ *such that* $\vdash \bar{S}' <: \bar{D}[\bar{S}'/\bar{X}]$ *we have that* $\vdash M[\bar{S}'/\bar{X}] \to N[\bar{S}'/\bar{X}]$.

The theorem states that for each reduction step involving a term M containing types $\bar{S}$, the type information can be abstracted out (into $\bar{X}$) and replaced with any other types $\bar{S}'$ which satisfy the constraints. Note that this is true of every step of reduction — at each step, the types can be replaced with any other type which satisfy the constraints.

7 Conclusion

In this paper, we have studied the incorporation of generic types in aspect languages in the context of both models of parametric polymorphism: the type erasure semantics and the type carrying semantics. Our study has accommodated the subtleties of the interaction of classes, polymorphism and aspects; advice complicates the notion of return type and complicates the type erasure semantics used by Generic Java.

Our pointcut model is simple: we only consider execution pointcuts and our pointcut logic is positive, i.e. no negation. We have argued that genericity enables us to recover several important uses of negation.

We demonstrate that generic advice is useful, even in the presence of nongeneric base classes. In general, we have argued that uses of augmentation and narrowing advice fall inside our framework.

This paper proves the fundamental properties of a static semantics: namely that well typing is preserved by reduction and that well typed programs can always make progress. We also describe conditions under which reduction is independent of type annotations. Our contribution is timely, as full source-level support for the generic features of Java 1.5 is just now available in AspectJ.

In future work, we intend to explore richer pointcut logics including those already present in AspectJ, such as `call` and `cflow`. We also intend to explore more expressive type rules for wildcards. For example, PC-WILD that may be more restrictive than necessary, by disallowing clearly sound examples such as the following.

$$\texttt{advice}\, a\langle X, Y\rangle\, Y\, \texttt{(int}\, x\texttt{)}\, \texttt{:exe}\, Y\, X\, \texttt{.*(int)}\{\texttt{proceed(}x+1\texttt{)}\}$$

We have also not addressed contravariant argument types in this paper. Consider a variant of the example from section 2: a class and an aspect intended to work on all subclasses.

```
class D {
  public String m(String s) { return "D"++s; }
}
aspect C<T> {
  T around(): execution(* D.m(String s)) {
```

```
    // what type may this code assume for s??
} }
```

Naively, if the aspect `C` has to work in a typesafe way in the presence of contravariant arguments, the only type assumption that the body of the aspect can make about the argument is `Object`. Clearly, this is quite restrictive, and the investigation of design issues to ease this impediment is left to future work.

In this (already long!) paper, we have not addressed the issues of weaving. The weaving algorithm translates the aspect-based programs of AFGJ into programs in the class-based FGJ. This algorithm is not novel: the untyped version of the algorithm is closely modeled on that used by AspectJ. In our earlier work on an untyped aspect calculus [54], we have analyzed such an algorithm formally for a calculus with more features (e.g. inner classes, state, call and execution pointcuts). Since weaving is primarily a direct reflection of aspect dynamics into object dynamics, we believe that weaving preserves typeability of programs by mapping well typed aspect programs to well typed class based programs. The formal exploration of this point is left to future work.

Acknowledgment

We wish to thank Glenn Bruns for many useful discussions during both the inception and elaboration of this work. We also wish to thank Steve Rogers and the anonymous reviewers of this and an earlier version of this report for several useful comments.

References

[1] L. Bergmans, Composing concurrent objects - applying composition filters for the development and reuse of concurrent object-oriented programs, Ph.d. thesis, University of Twente, http://wwwhome.cs.utwente.nl/~bergmans/phd.htm (1994).

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, Lecture Notes in Computer Science 2072 (2001) 327–355.

[3] H. Ossher, P. Tarr, Multi-dimensional separation of concerns and the hyperspace approach, in: Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, 2001.

[4] K. J. Lieberherr, Adaptive Object-Oriented Software: The Demeter method with propagation patterns, PWS Publishing Company, 1996.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: European Conference on Object-Oriented Programming (ECOOP), 1997.

[6] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa, Abstracting object-interactions using composition-filters, in: In object-based distributed processing, LNCS, 1993.

[7] R. Filman, D. Friedman, Aspect-oriented programming is quantification and obliviousness (2000).

[8] P. L. Tarr, H. Ossher, S. M. S. Jr., Hyper/J: multi-dimensional separation of concerns for java., in: ICSE, ACM, 2002, pp. 689–690.

[9] P. L. Tarr, H. Ossher, Hyper/J: Multi-dimensional separation of concerns for Java., in: ICSE, IEEE Computer Society, 2001, pp. 729–730.

[10] G. Kiczales, Y. Coady, Aspectc, http://www.cs.ubc.ca/labs/spl/projects/aspectc.html (2001).

[11] Y. Coady, G. Kiczales, M. J. Feeley, G. Smolyn, Using aspectc to improve the modularity of path-specific customization in operating system code., in: ESEC / SIGSOFT FSE, 2001, pp. 88–98.

[12] D. B. Tucker, S. Krishnamurthi, Pointcuts and advice in higher-order languages., in: AOSD, 2003, pp. 158–167.

[13] D. Walker, S. Zdancewic, J. Ligatti, A theory of aspects, in: Conference Record of ICFP 03: The ACM SIGPLAN International Conference on Functional Programming, 2003.

[14] G. C. Murphy, R. J. Walker, E. L. A. Baniassad, M. P. Robillard, A. Lai, M. Kersten, Does aspect-oriented programming work?, Commun. ACM 44 (10) (2001) 75–77.

[15] S. Matsuoka, A. Yonezawa, Analysis of Inheritance anomaly in Object-oriented concurrent programming languages, in: G. Agha, P. Wegner, A. Yonezawa (Eds.), Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993, pp. 107–150.

[16] C. V. Lopes, D: A language framework for distributed programming, Ph.d. thesis, Northestern University, ftp://ftp.ccs.neu.edu/pub/people/lieber/theses/lopes/dissertation.pdf (1997).

[17] M. Mezini, K. Ostermann, Variability management with feature-oriented programming and aspects, in: Proceedings of the 12th ACM SIGSOFT symposium on Foundations of software engineering, ACM Press, 2004, pp. 127–136.

[18] M. Mezini, K. Ostermann, Conquering aspects with Caesar., in: AOSD, 2003, pp. 90–99.

[19] A. M. Colyer, A. Clement, Large-scale AOSD for middleware., in: G. C. Murphy, K. J. Lieberherr (Eds.), AOSD, ACM, 2004, pp. 56–65.

[20] R. Khaled, J. Noble, R. Biddle, InspectJ: Program monitoring for visualisation using AspectJ., in: ACSC, 2003, pp. 359–368.

[21] C. Bockisch, M. Haupt, M. Mezini, K. Ostermann, Virtual machine support for dynamic join points., in: G. C. Murphy, K. J. Lieberherr (Eds.), AOSD, ACM, 2004, pp. 83–92.

[22] M. Eichberg, M. Mezini, K. Ostermann, Pointcuts as functional queries., in: APLAS, 2004, pp. 366–381.

[23] M. Nishizawa, S. Chiba, M. Tatsubori, Remote pointcut: a language construct for distributed AOP., in: G. C. Murphy, K. J. Lieberherr (Eds.), AOSD, ACM, 2004, pp. 7–15.

[24] S. Chiba, K. Nakagawa, Josh: an open Aspectj-like language., in: AOSD, 2004, pp. 102–111.

[25] K. D. Volder, T. D'Hondt, Aspect-Orientated Logic Meta-programming., in: P. Cointe (Ed.), Reflection, Vol. 1616 of Lecture Notes in Computer Science, Springer, 1999, pp. 250–272.

[26] B. D. Win, V. Shah, W. Joosen, R. Bodkin, AOSDSEC: AOSD Technology for Application-Level Security, http://www.cs.kuleuven.ac.be/~distrinet/events/aosdsec/ (2004).

[27] F. B. Schneider, J. G. Morrisett, R. Harper, A Language-based Approach to Security., in: R. Wilhelm (Ed.), Informatics, Vol. 2000 of Lecture Notes in Computer Science, Springer, 2001, pp. 86–101.

[28] O. Nierstrasz, Regular types for Active Objects., in: OOPSLA, 1993, pp. 1–15.

[29] G. Leavens, Report on the FOAL 2002 workshop, http://www.seas.upenn.edu/~sweirich/types/archive/1999-2003/msg01029.html (2002).

[30] K. J. Lieberherr, D. Lorenz, J. Ovlinger, Aspectual collaborations – combining modules and aspects, The Computer Journal 46 (5) (2003) 542–565, http://www.ccs.neu.edu/research/demeter/papers/ac-aspectj-hyperj.

[31] D. S. Dantas, D. Walker, G. Washburn, S. Weirich, Polyaml: a polymorphic aspect-oriented functional programming language, in: ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming, ACM Press, New York, NY, USA, 2005, pp. 306–319.

[32] H. Masuhara, H. Tatsuzawa, A. Yonezawa, Aspectual caml: an aspect-oriented functional language, in: ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming, ACM Press, New York, NY, USA, 2005, pp. 320–330.

[33] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler, Making the Future Safe for the Past: Adding Genericity to the Java programming language., in: OOPSLA, 1998, pp. 183–200.

[34] M. Odersky, P. Wadler, Pizza into Java: Translating theory into practice., in: POPL, 1997, pp. 146–159.

[35] A. Kennedy, D. Syme, Transposing f to c#: expressivity of parametric polymorphism in an object-oriented language., Concurrency - Practice and Experience 16 (7) (2004) 707–733.

[36] M. Abadi, L. Cardelli, A Theory of Objects, Springer Verlag, 1996.

[37] J. Saltzer, M. Schroeder, The protection of information in computer systems., in: IEEE, Vol. 9(63), 1975.

[38] A. P. Black, Object-Oriented languages: The Next Generation., ACM Comput. Surv. 28 (4es) (1996) 149.

[39] C. Clifton, G. T. Leavens, Obliviousness, modular reasoning, and the behavioral subtyping analogy, tR #03-01a (Jan 2003).

[40] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ., ACM Trans. Program. Lang. Syst. 23 (3) (2001) 396–450.

[41] A. K. Wright, M. Felleisen, A syntactic approach to type soundness, Information and Computation 115 (1) (1994) 38–94.
URL citeseer.ist.psu.edu/wright92syntactic.html

[42] K. B. Bruce, L. Cardelli, B. C. Pierce, Comparing object encodings, Information and Computation 155, an extended abstract appeared in Proceedings of TACS '97, LNCS 1281, Springer-Verlag, pp. 415-438.

[43] K. B. Bruce, A. Fiech, L. Petersen, Subtyping is not a good "match" for object-oriented languages, in: European Conference on Object-Oriented Programming (ECOOP), 1997.

[44] K. B. Bruce, A. Fiech, A. Schuett, R. van Gent, A type-safe polymorphic object-oriented language, in: European Conference on Object-Oriented Programming (ECOOP), 1995.

[45] K. Fisher, J. Reppy, J. G. Riecke, A calculus for compiling and linking classes, in: European Conference on Object-Oriented Programming (ECOOP), 2000.

[46] V. Bono, L. Liquori, A subtyping for the Fisher-Honsell-Mitchell lambda calculus of objects., in: CSL, 1994, pp. 16–30.

[47] M. Flatt, S. Krishnamurthi, M. Felleisen, Classes and Mixins, in: ACM Symposium on Principles of Programming Languages (POPL), 1998, pp. 171–183.

[48] S. Drossopoulou, S. Eisenbach, S. Khurshid, Is the Java type system sound?, Theory and Practice of Object Systems 5 (11) (1999) 3–24.

[49] G. Bierman, M. Parkinson, A. Pitts, An imperative core calculus for Java and Java with effects, Tech. Rep. 563, University of Cambridge Computer Laboratory (Apr. 2003).

[50] D. Yu, A. Kennedy, D. Syme, Formalization of generics for the .Net common language runtime, in: POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press, 2004, pp. 39–51.

[51] A. Igarashi, B. C. Pierce, On Inner classes., Inf. Comput. 177 (1) (2002) 56–89.

[52] M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, N. Gafter, Adding wildcards to the Java programming language, in: Proceedings of the 2004 ACM symposium on Applied computing, ACM Press, 2004, pp. 1289–1296.

[53] M. C. Rinard, A. Salcianu, S. Bugrara, A classification system and analysis for aspect-oriented programs., in: R. N. Taylor, M. B. Dwyer (Eds.), SIGSOFT FSE, ACM, 2004, pp. 147–158.

[54] R. Jagadeesan, A. Jeffrey, J. Riely, A calculus of untyped Aspect-oriented programs., in: L. Cardelli (Ed.), ECOOP, Vol. 2743 of Lecture Notes in Computer Science, Springer, 2003, pp. 54–73.

[55] P. Wadler, List comprehensions, in: The Implementation of Functional Programming Languages, Prentice Hall, 1987.

A AFGJ Summary

For reference, we accumulate those definitions relevant to the AFGJ type carrying semantics that are spread between Section 3 and Sections 4 and 5.

NAMES AND VARIABLES

a, b	*Advice names*
c, d	*Class names (*`Object` *reserved)*
f, g, h	*Field names*
ℓ	*Method names*
x	*Term Variables (*`this`, `target` *reserved)*
X, Y, Z, W	*Type Variables*

SYNTAX

$C,D,E ::= c\langle\bar{T}\rangle$	*Non-variable Type*
$P,Q,R,S,T,U,V ::= X \mid C$	*Types*
$A,B ::= a\langle\bar{T}\rangle$	*Advice Application*
$O ::= \texttt{new}\ C(\bar{O})$	*Values*
$M,N,L ::=$	*Terms*
$\quad x$	*Term Variable*
$\quad M.f$	*Field Access*
$\quad M.\ell\langle\bar{T}\rangle(\bar{N})$	*Method Call*
$\quad M.\ell\langle\bar{T}\rangle[\bar{A}](\bar{N})$	*Advised Call*
$\quad \texttt{proceed}(\bar{N})$	*Proceed Call*
$\quad \texttt{new}\ C(\bar{N})$	*Object*
$\mathscr{E} ::=$	*Evaluation Contexts*
$\quad [].f$	*Field Access*
$\quad [].\ell\langle\bar{V}\rangle(\bar{N})$	*Method Call Target*
$\quad [].\ell\langle\bar{T}\rangle[\bar{A}](\bar{N})$	*Advised Call Target*
$\quad M.\ell\langle\bar{V}\rangle(\bar{N},[],\bar{N}')$	*Method Call Argument*
$\quad M.\ell\langle\bar{T}\rangle[\bar{A}](\bar{N},[],\bar{N}')$	*Advised Call Argument*
$\quad \texttt{proceed}(\bar{N},[],\bar{N}')$	*Proceed Call Argument*
$\quad \texttt{new}\ C(\bar{N},[],\bar{N}')$	*Object*
$\mathscr{D} ::=$	*Top Level Declarations*
$\quad \texttt{class}\ c\langle\bar{X}\triangleleft\bar{C}\rangle\triangleleft D\{\bar{T}\ \bar{f};\ \kappa\ \bar{\mu}\}$	*Class Declaration*
$\quad \texttt{advice}\ a\langle\bar{X}\triangleleft\bar{C}\rangle\ R(\bar{P}\ \bar{x}) : \phi\{M\}$	*Advice Declaration*
$\kappa ::= c(\bar{T}\ \bar{f})\{\texttt{super}(\bar{g});\ \texttt{this}.\bar{h}=\bar{h};\}$	*Constructor Declarations*
$\mu ::= \langle\bar{X}\triangleleft\bar{C}\rangle R\ \ell(\bar{P}\ \bar{x})\{M\}$	*Method Declarations*
$\phi,\psi,\rho ::=$	*Pointcuts*
$\quad \texttt{exe}\ R\ T.\ell\langle\bar{V}\rangle(\bar{P})$	*Method Execution*
$\quad \texttt{exe}\ R\ T.\ell\langle\bar{V}\rangle(\bar{P},*)$	*Vararg Method Execution*
$\quad \texttt{exe}\ R\ T.*(\bar{P})$	*Wildcard Execution*
$\quad \texttt{exe}\ R\ T.*(\bar{P},*)$	*Vararg Wildcard Execution*
$\quad \phi\ \texttt{\&\&}\ \psi$	*And*
$\quad \phi\ \texttt{||}\ \psi$	*Or*
$\quad \texttt{false}$	*False*
$\quad \texttt{true}$	*True*
$\Delta ::= \bullet \mid \Delta, X\triangleleft C$	*Type Environment*
$\Gamma ::= \bullet \mid \Gamma, T\ x$	*Term Environment*
$\quad \Gamma, R\ \texttt{proceed}(\bar{P})$	*Proceed*

EVALUATION $(M \to M')$

(EVAL-LOOKUP)
$$\frac{[\bar{A}] = \left[a\langle \bar{U} \rangle \;\middle|\; \begin{array}{l} \mathscr{D} \ni \texttt{advice}\ a\ \cdots \\ \vdash C.\ell\langle \bar{V} \rangle\ \mathsf{advised\ by}\ a\langle \bar{U} \rangle \end{array} \right]}{M.\ell\langle \bar{V} \rangle\,(\bar{N}) \to M.\ell\langle \bar{V} \rangle\,[\bar{A}]\,(\bar{N})}\ M = \texttt{new}\ C(\cdots)$$

(EVAL-FIELD)
$$\frac{\mathsf{fields}(C) = \bar{f}}{\texttt{new}\ C(\bar{N}).f_i \to N_i}$$

(EVAL-METHOD)
$$\frac{\mathsf{meth}(C.\ell) = \langle \bar{X} \rangle\,(\bar{x})\,\{L\}}{M.\ell\langle \bar{V} \rangle\,[]\,(\bar{N}) \to L[{}^{\bar{V}}/_{\bar{X}}, {}^{M}/_{\texttt{this}}, {}^{\bar{N}}/_{\bar{x}}]}\ M = \texttt{new}\ C(\cdots)$$

(EVAL-CONTEXT)
$$\frac{M \to M'}{\mathscr{E}[M] \to \mathscr{E}[M']}$$

(EVAL-ADVICE)
$$\frac{\mathscr{D} \ni \texttt{advice}\ a\langle \bar{X} \rangle\ (\bar{x}) \cdots \{L\}}{M.\ell\langle \bar{V} \rangle\,[a\langle \bar{U} \rangle, \bar{A}]\,(\bar{N}, \bar{N}') \to L[{}^{\bar{U}}/_{\bar{X}}, {}^{M}/_{\texttt{target}}, {}^{\bar{N}}/_{\bar{x}}, {}^{M.\ell\langle \bar{V} \rangle\,[\bar{A}]\,(\bar{N}')}/_{\texttt{proceed}}]}$$

ENVIRONMENT TYPING $(\Delta; \Gamma \vdash \mathsf{ok})$

(ENV-EMPTY)
$$\frac{\Delta \vdash \mathsf{ok}}{\Delta; \bullet \vdash \mathsf{ok}}$$

(ENV-TERM-VAR)
$$\frac{\Delta; \Gamma \vdash \mathsf{ok} \quad \Delta; \Gamma \vdash T \quad x \notin \mathsf{dom}(\Gamma)}{\Delta; \Gamma, T\ x \vdash \mathsf{ok}}$$

(ENV-PROCEED)
$$\frac{\Delta; \Gamma \vdash \mathsf{ok} \quad \Delta; \Gamma \vdash \bar{P}, R \quad \texttt{proceed} \notin \mathsf{dom}(\Gamma)}{\Delta; \Gamma, R\ \texttt{proceed}(\bar{P}) \vdash \mathsf{ok}}$$

DECLARATION TYPING $(\vdash \mathscr{D})$

(DEC-ADVICE)
$$\frac{\begin{array}{l} \phi \vDash \mathsf{exe}\ R\ T.*(\bar{P}, *) \\ \mathsf{fv}(\phi) = \mathsf{dcfv}(\phi) = \{\bar{Y}\} \\ \bar{Y} \lhd \bar{E} \vdash T, \bar{E}, \bar{P}, R \\ \bar{Y} \lhd \bar{E}; \bar{P}\ \bar{x}, T\ \texttt{target}, R\ \texttt{proceed}(\bar{P}) \vdash M : R' \\ \bar{Y} \lhd \bar{E} \vdash R' <: R \end{array}}{\vdash \texttt{advice}\ a\langle \bar{Y} \lhd \bar{E} \rangle\ R(\bar{P}\ \bar{x}) : \phi\{M\}}$$

(DEC-CLASS)
$$\frac{\begin{array}{l} \bar{X} \lhd \bar{C} \vdash \bar{C}, D, \bar{T} \\ \mathsf{fields}(D) = \bar{S}\ \bar{g} \\ \bar{X} \lhd \bar{C}; \bar{S}\ \bar{g}, \bar{T}\ \bar{f} \vdash \mathsf{ok} \\ \vdash \bar{\mu} : \mathsf{ok\ in}\ c\langle \bar{X} \lhd \bar{C} \rangle \lhd D \\ \kappa = c(\bar{S}\ \bar{g}, \bar{T}\ \bar{f})\{\texttt{super}(\bar{g})\,;\ \texttt{this}.\bar{f} = \bar{f}\,;\} \end{array}}{\vdash \texttt{class}\ c\langle \bar{X} \lhd \bar{C} \rangle \lhd D\,\{\bar{T}\ \bar{f};\ \kappa\ \bar{\mu}\}}$$

TERM TYPING $(\Delta; \Gamma \vdash M : T)$

(TERM-FIELD)
$$\frac{\Delta; \Gamma \vdash M : T \quad \Delta \vdash \mathsf{fields}(T) = \bar{S}\ \bar{f}}{\Delta; \Gamma \vdash M.f_i : S_i}$$

(TERM-VAR)
$$\frac{\Delta; \Gamma \vdash \mathsf{ok} \quad \Gamma(x) = T}{\Delta; \Gamma \vdash x : T}$$

(TERM-ADVISED)
$$\frac{\Delta \vdash C.\ell\langle \bar{V} \rangle\ \mathsf{advised\ by}\ \bar{A} \quad \Delta; \Gamma \vdash M.\ell\langle \bar{V} \rangle\,(\bar{N}) : R}{\Delta; \Gamma \vdash M.\ell\langle \bar{V} \rangle\,[\bar{A}]\,(\bar{N}) : R}\ M = \texttt{new}\ C(\cdots)$$

(TERM-OBJECT)
$$\frac{\Delta; \Gamma \vdash \mathsf{ok} \quad \Delta \vdash C \quad \vdash \mathsf{fields}(C) = \bar{S}\ \bar{f} \quad \Delta; \Gamma \vdash \bar{N} : \bar{S}' \quad \Delta \vdash \bar{S}' <: \bar{S}}{\Delta; \Gamma \vdash \texttt{new}\ C(\bar{N}) : C}$$

(TERM-PROCEED)
$$\frac{\Delta; \Gamma \vdash \mathsf{ok} \quad \Gamma(\texttt{proceed}) = R(\bar{P}) \quad \Delta; \Gamma \vdash \bar{N} : \bar{P}' \quad \Delta \vdash \bar{P}' <: \bar{P}}{\Delta; \Gamma \vdash \texttt{proceed}(\bar{N}) : R}$$

(TERM-METHOD)
$$\frac{\Delta \vdash \bar{V} \quad \Delta \vdash \mathsf{meth}(T.\ell) = \langle \bar{Y} \lhd \bar{E} \rangle R(\bar{P}) \cdots \quad \Delta; \Gamma \vdash (M, \bar{N}) : (T, \bar{P}') \quad \Delta \vdash (\bar{V}, \bar{P}') <: (\bar{E}, \bar{P})[{}^{\bar{V}}/_{\bar{Y}}]}{\Delta; \Gamma \vdash M.\ell\langle \bar{V} \rangle\,(\bar{N}) : R}$$

B Proofs for FGJ

B.1 Preservation and Progress

The following lemmas state basic sanity requirements relating subtyping, well-formed types and lookup. In each case, the proofs follow by induction on the definition of subtyping. These lemmas are only used to prove properties of the dynamics and, thus, type variables are not necessary. We recall Assumption 3.2, that all declarations are well-typed.

LEMMA B.1 (SUPERTYPING PRESERVES WELL FORMED TYPE). *If* $\Delta \vdash \mathsf{ok}$ *and* $\Delta \vdash T$ *and* $\Delta \vdash T <: T'$ *then* $\Delta \vdash T'$.

LEMMA B.2 (SUBTYPING PRESERVES FIELD LOOKUP). *If* $\vdash T$ *and* $\vdash T <: T'$ *and* $\vdash \mathsf{fields}(T') = \bar{V}\ \bar{f}$ *then* $\vdash \mathsf{fields}(T) = \bar{V}\ \bar{f}, \bar{U}\ \bar{g}$ *for some* $\bar{g}$ *disjoint from* $\bar{f}$.

LEMMA B.3 (SUBTYPING PRESERVES METHOD LOOKUP). *If* $\vdash T$ *and* $\vdash T <: T'$ *and* $\vdash \mathsf{meth}(T'.\ell) = \langle \bar{Y} \triangleleft \bar{C} \rangle R(\bar{P}) \cdots$ *then* $\vdash \mathsf{meth}(T.\ell) = \langle \bar{Y} \triangleleft \bar{C} \rangle R'(\bar{P}) \cdots$ *and* $\vdash R' <: R$.

Further, note that terms can only be typed by well-formed environments

LEMMA B.4 (WELL FORMED TERM IMPLIES WELL FORMED ENVIRONMENT). *If* $\Delta; \Gamma \vdash M : T$ *then* $\Delta; \Gamma \vdash \mathsf{ok}$.

The following lemma states a sanity condition on method lookup.

LEMMA B.5 (METHOD LOOKUP PRESERVES TYPING). *If* $\vdash T$ *and* $\vdash \mathsf{meth}(T.\ell) = \langle \bar{Y} \triangleleft \bar{C} \rangle R(\bar{x}\ \bar{P})\{L\}$ *then* $\bar{Y} \triangleleft \bar{C}; \bar{P}\ \bar{x}, T\ \mathtt{this} \vdash L : R'$ *and* $\bar{Y} \triangleleft \bar{C} \vdash R' <: R$.

Proof. An induction on the derivation of $\vdash \mathsf{meth}(T.\ell) = \langle \bar{Y} \triangleleft \bar{C} \rangle R(\bar{x}\ \bar{P})\{L\}$, making use of the requirement that the global declaration set $\mathscr{D}$ is well-formed.

The following definition and lemma describe sanity conditions relating evaluation contexts and typing.

CONTEXT TYPING $(\vdash \mathscr{E} : T \to R)$

Define $\vdash \mathscr{E} : T \to R$ if for all M:
$(\exists T' . \vdash T' <: T$ and $\vdash M : T')$ implies $(\exists R' . \vdash R' <: R$ and $\vdash \mathscr{E}[M] : R')$.

LEMMA B.6 (CONTEXT TYPING). *If* $\vdash \mathscr{E}[M] : R$ *then* $\exists T . \vdash M : T$ *and* $\vdash \mathscr{E} : T \to R$.

Proof. An induction on $\vdash \mathscr{E}[M] : R$. □

The following lemma describes sanity conditions on substitutions. To state the lemma succinctly, we introduce a notation for all judgments that satisfy substitutivity. Term typing includes term variables and must be treated separately.

JUDGMENTS

$\mathscr{J}$ ::=	*Judgments*
T	*Well Formed Type*
$T <: S$	*Subtyping*
$\mathsf{fields}(C) = \bar{T}\ \bar{f}$	*Field Lookup*
$\mathsf{meth}(C.\ell) = \langle \bar{Y} \triangleleft \bar{E} \rangle R(\bar{P}\ \bar{x})\{M\}$	*Method Lookup*

LEMMA B.7 (SUBSTITUTIVITY). *(a) If* $\bar{Y} \triangleleft \bar{C} \vdash \mathscr{J}$ *and* $\vdash \bar{D}$ *and* $\vdash \bar{D} <: \bar{C}[\bar{D}/\bar{Y}]$ *then* $\vdash \mathscr{J}[\bar{D}/\bar{Y}]$. *(b) If* $\bar{Y} \triangleleft \bar{C}; \bar{P}\ \bar{x} \vdash M : S$ *and* $\vdash \bar{D}$ *and* $\vdash \bar{D} <: \bar{C}[\bar{D}/\bar{Y}]$ *and* $\vdash \bar{N} : \bar{Q}$ *and* $\vdash \bar{Q} <: \bar{P}[\bar{D}/\bar{Y}]$ *then* $\vdash M[\bar{D}/\bar{Y}, \bar{M}/\bar{x}] : T$ *and* $\vdash T <: S[\bar{D}/\bar{Y}]$.

Proof. Using Lemmas B.1–B.3, (a) follows by induction on $\bar{Y} \triangleleft \bar{C} \vdash \mathscr{J}$ and (b) follows by induction on $\bar{Y} \triangleleft \bar{C}; \bar{P}\ \bar{x} \vdash M : S$. The interesting cases for (b) are TERM-FIELD and TERM-METHOD which are similar. In the case of TERM-FIELD, we have $\bar{Y} \triangleleft \bar{C}; \bar{P}\ \bar{x} \vdash M.f_i : S_i$ from hypotheses $\bar{Y} \triangleleft \bar{C}; \bar{P}\ \bar{x} \vdash M : T$ and $\bar{Y} \triangleleft \bar{C} \vdash \mathsf{fields}(T) = \bar{S}\ \bar{f}$. If T is a ground type, then we proceed by induction using (a) and Lemma B.2. Otherwise, we have $T = Y_i$, and we must have used FIELD-VAR, so we have $\bar{Y} \triangleleft \bar{C} \vdash \mathsf{fields}(C_i) = \bar{S}\ \bar{f}$, and since $\vdash D_i <: C_i[\bar{D}/\bar{Y}]$, we proceed by induction using (a) and Lemma B.2. The case of TERM-METHOD is similar. □

As usual, weakening follows by induction on the judgment in the supposition.

LEMMA B.8 (WEAKENING). *(a) If* $\Delta \vdash \mathscr{J}$ *then* $\Delta, \Delta' \vdash \mathscr{J}$. *(b) If* $\Delta; \Gamma \vdash M : T$ *then* $\Delta, \Delta'; \Gamma, \Gamma' \vdash M : T$.

THEOREM (3.3 PRESERVATION). *If* $\vdash M : T$ *and* $M \to N$ *then* $\exists T'.\ \vdash T' <: T$ *and* $\vdash N : T'$.

Proof. By case analysis on $M \to N$. For EVAL-CONTEXT use Lemma B.6. For EVAL-METHOD use Lemmas B.5, and B.7. □

THEOREM (3.4 PROGRESS). *If* $\vdash M$ *then either* M *is a value or* $\exists N.\ M \to N$.

Proof. By induction on $\vdash M$. □

B.2 Weak Confluence

We now turn our attention to a proof of weak confluence, for which we first prove some technical lemmas relating reduction, evaluation contexts and substitution.

LEMMA B.9. *If* $M \to N$ *then there exists some* L *in which* x *occurs exactly once and* $M = L[M'/x]$ *and* $N = L[N'/x]$ *and* $M' \to N'$ *without use of* EVAL-CONTEXT.

Proof. An induction on the derivation of $M \to N$. □

LEMMA B.10. *If $M \to N$ then $L[M/x] \to^* L[N/x]$.*

Proof. An induction on L. □

LEMMA B.11. *If $M \to N$ and $M \to L$, both without use of* EVAL-CONTEXT*, then $N = L$.*

Proof. A case analysis of M. □

LEMMA B.12. *If* $\texttt{new}\ C(\bar{M}) \to^* N$ *then* $N = \texttt{new}\ C(\bar{N})$ *and* $\bar{M} \to^* \bar{N}$.

Proof. Follows from observing that the only rule which could have derived $\texttt{new}\ C(\bar{M}) \to N$ is EVAL-CONTEXT. □

LEMMA B.13. *If $L[M/x] \to N$ without use of* EVAL-CONTEXT *then either:*

(1) $L = x$,
(2) $N = N'[M/x]$ *and* $L[M'/x] \to N'[M'/x]$ *for any* M', *or*
(3) $M = \texttt{new}\ C(\bar{M})$ *and* $N = N'[\bar{M}/\bar{x}]$ *and* $L[\texttt{new}\ C(\bar{N})/x] \to N'[\bar{N}/\bar{x}]$ *for any* $\bar{N}$ *of the same length as* $\bar{M}$.

Proof. A case analysis of the derivation of $L[M/x] \to N$. □

THEOREM (3.5 WEAK CONFLUENCE). *If $M \to N_1$ and $M \to N_2$ then $\exists L \,.\, N_1 \to^* L$ and $N_2 \to^* L$.*

Proof. We first use Lemma B.9 to get that $M = L_i[M'_i/x_i]$ and $N_i = L_i[N'_i/x_i]$ and $M'_i \to N'_i$ without use of EVAL-CONTEXT and x_i occurs exactly once in L_i. We then have three cases to consider:

(1) If $L_1 = L[M'_2/x_2]$ and $L_2 = L[M'_1/x_1]$ then by Lemma B.10:

$$N_1 = L_1[N'_1/x_1] = L[M'_2/x_2][N'_1/x_1] = L[N'_1/x_1][M'_2/x_2] \to^* L[N'_1/x_1][N'_2/x_2] = L[N'_1/x_1, N'_2/x_2]$$

and symmetrically:

$$N_2 = L_2[N'_2/x_2] = L[M'_1/x_1][N'_2/x_2] \to^* L[N'_2/x_2][M'_1/x_1] = L[N'_1/x_1, N'_2/x_2]$$

as required.

(2) If $M'_1 = M_1[M'_2/x_2]$ and $L_2 = L_1[M_1/x_1]$ then by Lemma B.13 we have three subcases to consider:

(a) If $M_1 = x_2$ then $M'_1 = M'_2$, so by Lemma B.11 we have:

$$N_1 = L_1[N'_1/x_1] = L_1[x_2/x_1][N'_2/x_2] = L_1[M_1/x_1][N'_2/x_2] = L_2[N'_2/x_2] = N_2$$

(b) If $N_1' = L_1'[M_2'/x_2]$ and $M_1[N_2'/x_2] \to L_1'[N_2'/x_2]$ then by Lemma B.10 we have:

$$\begin{aligned} N_1 &= L_1[N_1'/x_1] = L_1[L_1'[M_2'/x_2]/x_1] \\ &= L_1[L_1'/x_1][M_2'/x_2] \to^* L_1[L_1'/x_1][N_2'/x_2] = L_1[L_1'[N_2'/x_2]/x_1] \end{aligned}$$

and similarly we have:

$$N_2 = L_2[N_2'/x_2] = L_1[M_1/x_1][N_2'/x_2] = L_1[M_1[N_2'/x_2]/x_1] \to^* L_1[L_1'[N_2'/x_2]/x_1]$$

as required.

(c) If $M_2' = \texttt{new}\ C(\bar{M}_2)$ (and hence, by Lemma B.12, $N_2' = \texttt{new}\ C(\bar{N}_2)$ where $\bar{M}_2 \to^* \bar{N}_2$) and $N_1' = L_1'[\bar{M}_2/\bar{x}_2]$ and $M_1[\texttt{new}\ C(\bar{N}_2)/x_2] \to L_1'[\bar{N}_2/\bar{x}_2]$, then we proceed as for the previous sub-case.

(3) If $M_2' = M_2[M_1'/x_1]$ and $L_1 = L_2[M_2/x_2]$ then we proceed as for the previous case.

Note that this proof only requires Lemmas B.9–B.13, and so can be re-used for other languages satisfying these properties. □

C Proofs for Type-Carrying AFGJ

C.1 Preservation and Progress

We give the proof of preservation; the proof of progress is much as before.

We begin with lemmas giving properties of pointcuts.

LEMMA C.1 (CUT). *If $\rho \vDash \phi$ and $\phi \vDash \psi$ then $\rho \vDash \psi$.*

Proof. An induction on the derivation of $\rho \vDash \phi$, with an inner induction on the derivation of $\phi \vDash \psi$. □

LEMMA C.2 (POINTCUT SUBSTITUTIVITY). *If $\phi \vDash \psi$ then $\phi[\bar{U}/\bar{X}] \vDash \psi[\bar{U}/\bar{X}]$.*

Proof. An induction on the proof of $\phi \vDash \psi$. □

The following lemma states that proceed substitutions are well behaved.

LEMMA C.3 (PROCEED SUBSTITUTIVITY). *If $R\ \texttt{proceed}(\bar{P}) \vdash L : S$ and $\bar{P}\ \bar{x} \vdash M.\ell\langle\bar{V}\rangle\ [\bar{A}]\ (\bar{x},\bar{N}) : R$ then $\vdash L[M.\ell\langle\bar{V}\rangle\ [\bar{A}]\ (\bar{N})/\texttt{proceed}] : S$.*

Proof. An induction on the judgment typing L. □

The proof of preservation proceeds, as before, by induction on the definition of evaluation. The most interesting case is EVAL-ADVICE, which we consider in the

rest of this section. Given

$$\vdash M.\ell\langle\bar{V}\rangle\,[a\langle\bar{U}\rangle,\bar{A}]\,(\bar{N},\bar{N}') : R \tag{C.1}$$

our goal is to show that for some R'

$$\vdash L[\bar{U}/\bar{X}, M/\mathtt{target}, \bar{N}/\bar{x}, M.\ell\langle\bar{V}\rangle\,[\bar{A}]\,(\bar{N}')/\mathtt{proceed}] : R'$$
$$\vdash R' <: R$$

where we may assume that the following holds.

$$\mathscr{D} \ni \mathtt{advice}\, a\langle\bar{X}\triangleleft\bar{E}\rangle\, S(\bar{P}\,\bar{x}) : \phi\{L\} \tag{C.2}$$

The typing of the advised call (C.1) must follow from TERM-ADVISED, therefore we must have

$$M = \mathtt{new}\ C(\cdots) \tag{C.3}$$
$$\vdash C.\ell\langle\bar{V}\rangle \text{ advised by } a\langle\bar{U}\rangle,\bar{A} \tag{C.4}$$
$$\vdash M.\ell\langle\bar{V}\rangle\,(\bar{N},\bar{N}') : R \tag{C.5}$$

From (C.4), $\vdash C.\ell\langle\bar{V}\rangle$ advised by $\bar{A}$. Applying TERM-ADVISED to this and (C.5) gives us

$$\vdash M.\ell\langle\bar{V}\rangle\,[\bar{A}]\,(\bar{N},\bar{N}') : R \tag{C.6}$$

Using Assumption 3.2, the declaration of a in (C.2). must be typed. The only applicable rule is DEC-ADVICE, therefore we must have

$$\phi \vDash \mathtt{exe}\, S\, C.*(\bar{P},*) \tag{C.7}$$
$$\bar{X}\triangleleft\bar{E};\bar{P}\,\bar{x},C\,\mathtt{target},S\,\mathtt{proceed}(\bar{P}) \vdash L : S' \tag{C.8}$$
$$\bar{X}\triangleleft\bar{E} \vdash S' <: S \tag{C.9}$$

Note from (C.3) and (C.5), using TERM-METHOD and Lemma B.4, that C must be well formed and thus must not contain type variables. From (C.7) and Lemma C.2 (Pointcut substitutivity), we have

$$\phi[\bar{U}/\bar{X}] \vDash \mathtt{exe}\, S[\bar{U}/\bar{X}]\, C.*(\bar{P}[\bar{U}/\bar{X}],*) \tag{C.10}$$

From (C.4), $\vdash C.\ell\langle\bar{V}\rangle$ advised by $a\langle\bar{U}\rangle$, therefore the premises in the definition of advised by must hold.

$$\vdash \bar{U} <: \bar{E}[\bar{U}/\bar{X}] \tag{C.11}$$
$$\vdash \mathsf{meth}(C.\ell) = \langle\bar{Y}\rangle S''(\bar{Q},\bar{Q}')\cdots \tag{C.12}$$
$$\mathtt{exe}\, S''[\bar{V}/\bar{Y}]\, C.\ell\langle\bar{V}\rangle\,(\bar{Q}[\bar{V}/\bar{Y}],\bar{Q}'[\bar{V}/\bar{Y}]) \vDash \phi[\bar{U}/\bar{X}] \tag{C.13}$$

Applying Lemma C.1 (Cut) to (C.10) and (C.13), we have

$$\mathtt{exe}\, S''[\bar{V}/\bar{Y}]\, C.\ell\langle\bar{V}\rangle\,(\bar{Q}[\bar{V}/\bar{Y}],\bar{Q}'[\bar{V}/\bar{Y}]) \vDash \mathtt{exe}\, S[\bar{U}/\bar{X}]\, C.*(\bar{P}[\bar{U}/\bar{X}],*)$$

and thus

$$\bar{Q}[\bar{V}/\bar{Y}] = \bar{P}[\bar{U}/\bar{X}] \text{ and } S''[\bar{V}/\bar{Y}] = S[\bar{U}/\bar{X}] \tag{C.14}$$

Applying TERM-METHOD twice to (C.5) and (C.12), we have

$$\bar{Q}[\bar{V}/\bar{Y}]\ \bar{x} \vdash M.\ell\langle\bar{V}\rangle\,(\bar{x},\bar{N}') : S''[\bar{V}/\bar{Y}]$$
$$\vdash \bar{N} : \bar{Q}[\bar{V}/\bar{Y}]$$

From this and (C.14) we have

$$\bar{P}[\bar{U}/\bar{X}]\ \bar{x} \vdash M.\ell\langle\bar{V}\rangle\,(\bar{x},\bar{N}') : S[\bar{U}/\bar{X}] \tag{C.15}$$
$$\vdash \bar{N} : \bar{P}[\bar{U}/\bar{X}] \tag{C.16}$$

Applying TERM-METHOD again, we have

$$\vdash M.\ell\langle\bar{V}\rangle\,(\bar{N},\bar{N}') : S[\bar{U}/\bar{X}]$$

and comparison with (C.5) yields

$$S[\bar{U}/\bar{X}] = R \tag{C.17}$$

Applying TERM-ADVISED to (C.15) yields

$$\bar{P}[\bar{U}/\bar{X}]\ \bar{x} \vdash M.\ell\langle\bar{V}\rangle\,[\bar{A}]\,(\bar{x},\bar{N}') : S[\bar{U}/\bar{X}] \tag{C.18}$$

Using (C.11), we can apply a type substitution to (C.8) and (C.9). Further using (C.17) to replace $S[\bar{U}/\bar{X}]$ with R yields:

$$P[\bar{U}/\bar{X}]\ \bar{x}, C\ \texttt{target}, R\ \texttt{proceed}(\bar{P}[\bar{U}/\bar{X}]) \vdash L[\bar{U}/\bar{X}] : S'[\bar{U}/\bar{X}] \tag{C.19}$$
$$\vdash S'[\bar{U}/\bar{X}] <: R \tag{C.20}$$

Applying OBJECT and weakening to (C.3) we have

$$\bar{P}[\bar{U}/\bar{X}]\ \bar{x} \vdash M : C$$

Using this and (C.16), we can apply substitutivity (Lemma B.7) to (C.19), yielding

$$R\ \texttt{proceed}(\bar{P}[\bar{U}/\bar{X}]) \vdash L[\bar{U}/\bar{X}, M/\texttt{target}, \bar{N}/\bar{x},] : S'[\bar{U}/\bar{X}]$$

Finally, we can use (C.18) to apply pointcut substitutivity (Lemma C.3), yielding

$$\vdash L[\bar{U}/\bar{X}, M/\texttt{target}, \bar{N}/\bar{x}, {}^{M.\ell\langle\bar{V}\rangle\,[\bar{A}]\,(\bar{N}')}/\texttt{proceed}] : S'[\bar{U}/\bar{X}]$$

This, combined with (C.20) fulfills our obligation.

C.2 Weak Confluence

For confluence, the following lemma is sufficient to establish Proposition 5.1 (Deterministic advice lookup), which in turn establishes Theorem 5.7 (AFGJ Confluence).

LEMMA C.4. *If* $\mathsf{dcfv}(\phi) = \{\bar{X}\}$ *and* $\mathtt{exe}\ R\ T\ .\ \ell\langle\bar{V}\rangle\,(\bar{P}) \vDash \phi[\bar{U}/\bar{X}, \bar{U'}/\bar{X'}]$ *and* $\mathtt{exe}\ R\ T\ .\ \ell\langle\bar{V}\rangle\,(\bar{P}) \vDash \phi[\bar{R}/\bar{X}, \bar{R'}/\bar{X'}]$ *then* $\bar{U} = \bar{R}$.

Proof. By induction on ϕ. □

PROPOSITION (5.1 DETERMINISTIC ADVICE LOOKUP). *If* $\vdash T\ .\ \ell\langle\bar{V}\rangle$ advised by $a\langle\bar{U}\rangle$ *and* $\vdash T\ .\ \ell\langle\bar{V}\rangle$ advised by $a\langle\bar{U'}\rangle$ *then* $\bar{U} = \bar{U'}$.

Proof. Follows directly from Lemma C.4. □

THEOREM (5.7 AFGJ WEAK CONFLUENCE). *If* $\vdash M : T$, *for some* T, *and* $M \to N_1$ *and* $M \to N_2$ *then* $\exists L\,.\,N_1 \to^* L$ *and* $N_2 \to^* L$.

Proof. Follows the same structure as the proof of Theorem 3.5. The only tricky case in establishing the Lemmas B.9–B.13, is Lemma B.11, which makes use of Proposition 5.1 in the case of EVAL-ADVICE. □

D Proofs for Type-Erased AFGJ

In this section, we prove Theorem 6.5 (Parametricity of Reduction). We first prove substitutivity of advice lookup.

PROPOSITION D.1 (SUBSTITUTIVITY OF ADVICE LOOKUP). *Suppose* $\bar{X}\triangleleft\bar{D} \Vdash \mathsf{ok}$ *and* $\bar{X}\triangleleft\bar{D} \Vdash c\langle\bar{S}\rangle$ *and* $\vdash \bar{T}$ *and* $\vdash \bar{T} <: \bar{D}[\bar{T}/\bar{X}]$. *Then* $\bar{X}\triangleleft\bar{D} \vdash c\langle\bar{S}\rangle\,.\,\ell\langle\bar{V}\rangle$ advised by $a\langle\bar{Q}\rangle$ *implies* $\vdash c\langle\bar{S}[\bar{T}/\bar{X}]\rangle\,.\,\ell\langle\bar{V}[\bar{T}/\bar{X}]\rangle$ advised by $a\langle\bar{Q}[\bar{T}/\bar{X}]\rangle$.

Proof. From the definition of advice lookup, we have:

$$\begin{array}{l}\mathscr{D} \ni \mathtt{advice}\ a\langle\bar{Y}\triangleleft\bar{E}\rangle : \phi\ \cdots \\ \bar{X}\triangleleft\bar{D} \Vdash R\ c\langle\bar{S}\rangle\,.\,\ell\langle\bar{V}\rangle\,(\bar{P}) \\ \mathtt{exe}\ R\ c\langle\bar{S}\rangle\,.\,\ell\langle\bar{V}\rangle\,(\bar{P}) \vDash \phi[\bar{Q}/\bar{Y}] \\ \bar{X}\triangleleft\bar{D} \vdash \bar{Q} <: \bar{E}[\bar{Q}/\bar{Y}]\end{array}$$

and so by Lemma C.2 (Substitutivity of pointcut satisfaction):

$$(\mathtt{exe}\ R\ c\langle\bar{S}\rangle\,.\,\ell\langle\bar{V}\rangle\,(\bar{P}))[\bar{T}/\bar{X}] \vDash \phi[\bar{Q}/\bar{Y}][\bar{T}/\bar{X}]$$

so by Lemma B.7 (Substitutivity of method lookup, of well formed types and of subtyping):

$$\vdash R[\bar{T}/\bar{X}]\ c\langle \bar{S}[\bar{T}/\bar{X}]\rangle.\ell\langle \bar{V}[\bar{T}/\bar{X}]\rangle(\bar{P}[\bar{T}/\bar{X}])$$
$$\vdash \bar{Q}[\bar{T}/\bar{X}]$$
$$\vdash \bar{Q}[\bar{T}/\bar{X}] <: \bar{E}[\bar{Q}[\bar{T}/\bar{X}]/\bar{Y}]$$

so by the definition of advice lookup:

$$\vdash c\langle \bar{S}[\bar{T}/\bar{X}]\rangle.\ell\langle \bar{V}[\bar{T}/\bar{X}]\rangle \text{ advised by } a\langle \bar{Q}[\bar{T}/\bar{X}]\rangle$$

as required. □

We now prove inverse substitutivity of advice lookup, which allows us to deduce the erasure theorem. The proof of inverse substitutivity requires similar results for subtyping and pointcut satisfaction. We make use of an auxiliary 'well formed method typing' judgment.

WELL FORMED METHOD TYPING $(\Delta \vdash R\,T.\ell\langle \bar{V}\rangle(\bar{P}))$

$$\frac{\Delta \vdash T, \bar{V} \quad \Delta \vdash \mathsf{meth}(T.\ell) = \langle \bar{Y} \triangleleft \bar{C}\rangle R(\bar{P})\cdots \quad \Delta \vdash \bar{V} <: \bar{C}[\bar{V}/\bar{Y}]}{\Delta \vdash R[\bar{V}/\bar{Y}]\ T.\ell\langle \bar{V}\rangle(\bar{P}[\bar{V}/\bar{Y}])}$$

LEMMA D.2 (WELL FORMED RETURN TYPE). *If* $\Delta \vdash R\,T.\ell\langle \bar{V}\rangle(\bar{P})$ *then* $\Delta \vdash R$.

Proof. An induction on the derivation of $\Delta \vdash \mathsf{meth}(T.\ell) = \langle \bar{Y} \triangleleft \bar{C}\rangle R'(\bar{P})\cdots$.

LEMMA D.3 (MONOTONICITY OF METHOD LOOKUP). *If* $\Delta \vdash T <: T'$ *and* $\Delta \vdash R\,T.\ell\langle \bar{V}\rangle(\bar{P})$ *and* $\Delta \vdash \mathsf{meth}(T'.\ell) = \cdots$ *then* $\Delta \vdash R'\ T'.\ell\langle \bar{V}\rangle(\bar{P})$ *where* $\Delta \vdash R <: R'$.

Proof. An induction on the derivation of $\Delta \vdash T <: T'$.

LEMMA D.4 (INVERSE SUBSTITUTIVITY OF SUBTYPING). *Suppose* $\bar{X} \triangleleft \bar{D} \Vdash \mathsf{ok}$ *and* $\bar{X} \triangleleft \bar{D} \vdash c\langle \bar{S}\rangle$ *and* $\vdash \bar{T}$ *and* $\vdash \bar{T} <: \bar{D}[\bar{T}/\bar{X}]$. *Then* $\vdash c\langle \bar{S}[\bar{T}/\bar{X}]\rangle <: d\langle \bar{U}\rangle$ *implies* $\bar{X} \triangleleft \bar{D} \vdash c\langle \bar{S}\rangle <: d\langle \bar{Q}\rangle$ *and* $\bar{Q} = \bar{U}[\bar{T}/\bar{X}]$.

Proof. An induction on the derivation of $\vdash c\langle \bar{S}[\bar{T}/\bar{X}]\rangle <: d\langle \bar{U}\rangle$. □

LEMMA D.5 (ERASURE TYPING IMPLIES WELL FORMED ENVIRONMENT). *If* $\bar{Y} \triangleleft \bar{E} \Vdash \phi$ *and* $\bar{Y}$ *are distinct then* $\bar{Y} \triangleleft \bar{E} \vdash \mathsf{ok}$

LEMMA D.6 (INVERSE SUBSTITUTIVITY OF POINTCUT SATISFACTION). *Suppose* $\bar{X} \triangleleft \bar{D} \Vdash \mathsf{ok}$ *and* $\bar{X} \triangleleft \bar{D} \Vdash R\,c\langle \bar{S}\rangle.\ell\langle \bar{V}\rangle(\bar{P})$ *and* $\bar{Y} \triangleleft \bar{E} \Vdash \mathsf{ok}$ *and* $\bar{Y} \triangleleft \bar{E} \Vdash \phi$ *and* $\vdash \bar{T}$ *and* $\vdash \bar{T} <: \bar{D}[\bar{T}/\bar{X}]$ *and* $\vdash \bar{U}$ *and* $\vdash \bar{U} <: \bar{E}[\bar{U}/\bar{Y}]$. *Then*

$(\texttt{exe}\ R\ c\langle\bar{S}\rangle.\ell\langle\bar{V}\rangle(\bar{P}))[\bar{T}/\bar{X}] \vDash \phi[\bar{U}/\bar{Y}]$ *implies* $\bar{U} = \bar{Q}[\bar{T}/\bar{X}]$ *and* $\bar{X}\triangleleft\bar{D} \vdash \bar{Q}$ *and* $\bar{X}\triangleleft\bar{D} \vdash \bar{Q} <: \bar{E}[\bar{Q}/\bar{Y}]$ *and* $\texttt{exe}\ R\ c\langle\bar{S}\rangle.\ell\langle\bar{V}\rangle(\bar{P}) \vDash \phi[\bar{Q}/\bar{Y}]$.

Proof. An induction on the derivation of $\bar{Y}\triangleleft\bar{E} \Vdash \phi$. The interesting cases are conjunction and the base cases:

Case PC-AND

From PC-AND we have $\phi = \phi_1 \,\&\&\, \phi_2$ and $(\bar{Y}\triangleleft\bar{E}) = (\bar{Y}_1\triangleleft\bar{E}_1, \bar{Y}_2\triangleleft\bar{E}_2)$ where $\bar{Y}_i\triangleleft\bar{E}_i \Vdash \phi_i$ and $\bar{Y}_1$ and $\bar{Y}_2$ are disjoint. We use Lemma D.5 (Erasure typing implies well formed environment) to get that $\bar{Y}_i\triangleleft\bar{E}_i \Vdash \mathsf{ok}$, so we can use induction to find appropriate $\bar{Q}_i$ and define $\bar{Q} = (\bar{Q}_1, \bar{Q}_2)$: it is routine to verify that $\bar{Q}$ satisfies the required conditions. Note that this case relies on $\bar{Y}_1$ and $\bar{Y}_2$ being disjoint: if they were not, then the substitution $[\bar{Q}/\bar{Y}]$ would not be well-defined.

Case PC-EXE-CLASS

From PC-EXE-CLASS, we have:

$$\begin{array}{l}
\phi = \texttt{exe}\ R'\ d\langle\bar{Y}_1\rangle.\ell\langle\bar{Y}_2\rangle(\bar{P}') \\
\bar{Y}\triangleleft\bar{E} = \bar{Y}_1\triangleleft\bar{E}_1, \bar{Y}_2\triangleleft\bar{E}_2 \\
\mathscr{D} \ni \texttt{class}\ d\langle\bar{Y}_1\triangleleft\bar{Q}_1\rangle\cdots \\
\vdash \mathsf{meth}(d\langle\bar{Y}_1\rangle.\ell) = \langle\bar{Y}_2\triangleleft\bar{Q}_2\rangle R'(\bar{P}')\cdots \\
\bar{Y}_1\triangleleft\bar{Q}_1, \bar{Y}_2\triangleleft\bar{Q}_2 \vdash \bar{Q}_1, \bar{Q}_2 <: \bar{E}_1, \bar{E}_2
\end{array}$$

and we can split $\bar{U}$ into $\bar{U}_1, \bar{U}_2$ such that:

$$\vdash \bar{U}_1, \bar{U}_2 <: (\bar{E}_1, \bar{E}_2)[\bar{U}/\bar{Y}]$$

$(\texttt{exe}\ R\ c\langle\bar{S}\rangle.\ell\langle\bar{V}\rangle(\bar{P}))[\bar{T}/\bar{X}] \vDash \phi[\bar{U}/\bar{Y}]$ can only come from an axiom, in which case:

$$\begin{array}{l}
R[\bar{T}/\bar{X}] = R'[\bar{U}/\bar{Y}] \\
c = d \\
\bar{S}[\bar{T}/\bar{X}] = \bar{Y}_1[\bar{U}/\bar{Y}] = \bar{U}_1 \\
\bar{V}[\bar{T}/\bar{X}] = \bar{Y}_2[\bar{U}/\bar{Y}] = \bar{U}_2 \\
\bar{P}[\bar{T}/\bar{X}] = \bar{P}'[\bar{U}/\bar{Y}]
\end{array}$$

Since $\bar{X}\triangleleft\bar{D} \Vdash R\ c\langle\bar{S}\rangle.\ell\langle\bar{V}\rangle(\bar{P})$, and and using the above, we have:

$$\begin{array}{l}
\bar{X}\triangleleft\bar{D} \vdash \bar{S}, \bar{V} \\
\bar{X}\triangleleft\bar{D} \vdash \bar{S} <: \bar{Q}_1[\bar{S}/\bar{Y}_1] <: \bar{E}_1[\bar{S}/\bar{Y}_1] \\
\bar{X}\triangleleft\bar{D} \vdash \bar{V} <: \bar{Q}_2[\bar{S}/\bar{Y}_1][\bar{V}/\bar{Y}_2] <: \bar{E}_2[\bar{S}/\bar{Y}_1][\bar{V}/\bar{Y}_2] \\
R = R'[\bar{S}/\bar{Y}_1][\bar{V}/\bar{Y}_2] \\
\bar{P} = \bar{P}'[\bar{S}/\bar{Y}_1][\bar{V}/\bar{Y}_2]
\end{array}$$

Hence we can define:

$$\bar{Q} = \bar{S}, \bar{V}$$

and we have:

$$\begin{array}{l}
\bar{U} = \bar{U}_1, \bar{U}_2 = \bar{S}[\bar{T}/\bar{X}], \bar{V}[\bar{T}/\bar{X}] = \bar{Q}[\bar{T}/\bar{X}] \\
\bar{X} \triangleleft \bar{D} \vdash \bar{Q} \\
\bar{X} \triangleleft \bar{D} \vdash \bar{Q} <: \bar{E}[\bar{Q}/\bar{Y}] \\
\mathtt{exe}\ R'\ c\langle \bar{S} \rangle . \ell \langle \bar{V} \rangle (\bar{P}) \vDash (\mathtt{exe}\ R'\ c\langle Y_1 \rangle . \ell \langle Y_2 \rangle (\bar{P}'))[\bar{Q}/\bar{Y}] = \phi[\bar{Q}/\bar{Y}]
\end{array}$$

as required.

Case PC-EXE-VAR

From PC-EXE-VAR, we have:

$$\begin{array}{l}
\phi = \mathtt{exe}\ Y_4\ Y_1 . \ell \langle \bar{Y}_3 \rangle (\bar{P}') \\
\bar{Y} \triangleleft \bar{E} = Y_1 \triangleleft E_1, \bar{Y}_2 \triangleleft \bar{E}_2, \bar{Y}_3 \triangleleft \bar{E}_3, Y_4 \triangleleft E_4 \\
E_1 = d\langle \bar{Y}_2 \rangle \\
\mathscr{D} \ni \mathtt{class}\ d\langle \bar{Y}_2 \triangleleft \bar{E}_2 \rangle \cdots \\
\vdash \mathsf{meth}(d\langle \bar{Y}_2 \rangle . \ell) = \langle \bar{Y}_3 \triangleleft \bar{Q}_3 \rangle Q_4 (\bar{P}') \cdots \\
Y_1 \triangleleft d\langle \bar{Y}_2 \rangle, \bar{Y}_2 \triangleleft \bar{E}_2, \bar{Y}_3 \triangleleft \bar{Q}_3, Y_4 \triangleleft Q_4 \vdash \bar{Q}_3, Q_4 <: \bar{E}_3, E_4
\end{array}$$

and we can split $\bar{U}$ into $U_1, \bar{U}_2, \bar{U}_3, U_4$ such that:

$$\vdash U_1, \bar{U}_2, \bar{U}_3, U_4 <: (E_1, \bar{E}_2, \bar{E}_3, E_4)[\bar{U}/\bar{Y}]$$

$(\mathtt{exe}\ R\ c\langle \bar{S} \rangle . \ell \langle \bar{V} \rangle (\bar{P}))[\bar{T}/\bar{X}] \vDash \phi[\bar{U}/\bar{Y}]$ can only come from an axiom, in which case:

$$\begin{array}{l}
R[\bar{T}/\bar{X}] = Y_4[\bar{U}/\bar{Y}] = U_4 \\
c\langle \bar{S}[\bar{T}/\bar{X}] \rangle = Y_1[\bar{U}/\bar{Y}] = U_1 \\
\bar{V}[\bar{T}/\bar{X}] = \bar{Y}_3[\bar{U}/\bar{Y}] = \bar{U}_3 \\
\bar{P}[\bar{T}/\bar{X}] = \bar{P}'[\bar{U}/\bar{Y}]
\end{array}$$

We have:

$$\vdash c\langle \bar{S}[\bar{T}/\bar{X}] \rangle = U_1 <: E_1[\bar{U}/\bar{Y}] = d\langle \bar{U}_2 \rangle$$

so by Lemma D.4 (Inverse substitutivity of subtyping):

$$\begin{array}{l}
\bar{X} \triangleleft \bar{D} \vdash c\langle \bar{S} \rangle <: d\langle \bar{Q}_2 \rangle = \bar{E}_1[\bar{Q}_2/\bar{Y}_2] \\
\bar{Q}_2[\bar{T}/\bar{X}] = \bar{U}_2
\end{array}$$

and so by Lemma B.1 (Subtyping preserves well-formed type):

$$\bar{X} \triangleleft \bar{D} \vdash \bar{Q}_2$$
$$\bar{X} \triangleleft \bar{D} \vdash \bar{Q}_2 <: \bar{E}_2[\bar{Q}_2/\bar{Y}_2]$$

Since $\bar{X} \triangleleft \bar{D} \Vdash R\ c\langle \bar{S} \rangle . \ell \langle \bar{V} \rangle (\bar{P})$, we use Lemmas D.2 (Well formed return type) and D.3 (Monotonicity of method lookup): to get:

$$\bar{X} \triangleleft \bar{D} \vdash c\langle \bar{S} \rangle, \bar{V}, R$$
$$\bar{X} \triangleleft \bar{D} \vdash R'\ d\langle \bar{Q}_2 \rangle . \ell \langle \bar{V} \rangle (\bar{P})$$
$$\bar{X} \triangleleft \bar{D} \vdash R <: R'$$

and by definition of $\bar{X} \triangleleft \bar{D} \vdash R'\ d\langle \bar{Q}_2 \rangle . \ell \langle \bar{V} \rangle (\bar{P})$ we have:

$$\bar{X} \triangleleft \bar{D} \vdash \bar{V} <: \bar{Q}_3[\bar{Q}_2/\bar{Y}_2, \bar{V}/\bar{Y}_3] <: \bar{E}_3[\bar{Q}_2/\bar{Y}_2, \bar{V}/\bar{Y}_3]$$
$$R' = Q_4[\bar{Q}_2/\bar{Y}_2, \bar{V}/\bar{Y}_3] <: E_4[\bar{Q}_2/\bar{Y}_2, \bar{V}/\bar{Y}_3]$$
$$\bar{P} = \bar{P}'[\bar{Q}_2/\bar{Y}_2, \bar{V}/\bar{Y}_3]$$

Plugging the above together, we have:

$$\bar{X} \triangleleft \bar{D} \vdash c\langle \bar{S} \rangle, \bar{Q}_2, \bar{V}, R$$
$$\bar{X} \triangleleft \bar{D} \vdash c\langle \bar{S} \rangle, \bar{Q}_2, \bar{V}, R <: (E_1, \bar{E}_2, \bar{E}_3, E_4)[c\langle \bar{S} \rangle/Y_1, \bar{Q}_2/\bar{Y}_2, \bar{V}/\bar{Y}_3, R/Y_4]$$
$$\bar{P} = \bar{P}'[c\langle \bar{S} \rangle/Y_1, \bar{Q}_2/\bar{Y}_2, \bar{V}/\bar{Y}_3, R/Y_4]$$

so we can define:

$$\bar{Q} = c\langle \bar{S} \rangle, \bar{Q}_2, \bar{V}, R$$

and we have:

$$\bar{U} = U_1, \bar{U}_2, \bar{U}_3, U_4 = c\langle \bar{S}[\bar{T}/\bar{X}] \rangle, \bar{Q}_2[\bar{T}/\bar{X}], \bar{V}[\bar{T}/\bar{X}], R[\bar{T}/\bar{X}] = \bar{Q}[\bar{T}/\bar{X}]$$
$$\bar{X} \triangleleft \bar{D} \vdash \bar{Q}$$
$$\bar{X} \triangleleft \bar{D} \vdash \bar{Q} <: \bar{E}[\bar{Q}/\bar{Y}]$$
$$\texttt{exe}\ R\ c\langle \bar{S} \rangle . \ell \langle \bar{V} \rangle (\bar{P}) \vDash (\texttt{exe}\ Y_4\ Y_1 . \ell \langle \bar{Y}_3 \rangle (\bar{P}'))[\bar{Q}/\bar{Y}] = \phi[\bar{Q}/\bar{Y}]$$

as required. □

PROPOSITION D.7 (INVERSE SUBSTITUTIVITY OF ADVICE LOOKUP). *Suppose* $\bar{X} \triangleleft \bar{D} \Vdash \mathsf{ok}$ *and* $\bar{X} \triangleleft \bar{D} \Vdash R\ c\langle \bar{S} \rangle . \ell \langle \bar{V} \rangle (\bar{P})$ *and* $\vdash \bar{T}$ *and* $\vdash \bar{T} <: \bar{D}[\bar{T}/\bar{X}]$. *Then* $\vdash c\langle \bar{S}[\bar{T}/\bar{X}] \rangle . \ell \langle \bar{V}[\bar{T}/\bar{X}] \rangle$ advised by $a\langle \bar{U} \rangle$ *implies* $\bar{U} = \bar{Q}[\bar{T}/\bar{X}]$ *and* $\bar{X} \triangleleft \bar{D} \vdash c\langle \bar{S} \rangle . \ell \langle \bar{V} \rangle$ advised by $a\langle \bar{Q} \rangle$.

Proof. From the definition of advice lookup, we have:

$$\begin{array}{l}\mathscr{D} \ni \texttt{advice}\, a\langle \bar{Y} \triangleleft \bar{E} \rangle : \phi\ \cdots \\ (\texttt{exe}\, R\, c\langle \bar{S} \rangle . \ell \langle \bar{V} \rangle\, (\bar{P}))[\bar{T}/\bar{X}] \vDash \phi[\bar{U}/\bar{Y}] \\ \vdash \bar{U} \\ \vdash \bar{U} <: \bar{E}[\bar{U}/\bar{Y}]\end{array}$$

and since $\bar{X} \triangleleft \bar{D} \Vdash \mathsf{ok}$, we have by DEC-ADVICE and Lemma B.4 (Well formed term implies well formed environment):

$$\begin{array}{l}\bar{Y} \triangleleft \bar{E} \Vdash \mathsf{ok} \\ \bar{Y} \triangleleft \bar{E} \Vdash \phi\end{array}$$

and so by Lemma D.6 (Inverse substitutivity of pointcut satisfaction):

$$\begin{array}{l}\bar{U} = \bar{Q}[\bar{T}/\bar{X}] \\ \bar{X} \triangleleft \bar{D} \vdash \bar{Q} \\ \bar{X} \triangleleft \bar{D} \vdash \bar{Q} <: \bar{E}[\bar{Q}/\bar{Y}] \\ \texttt{exe}\, R\, c\langle \bar{S} \rangle . \ell \langle \bar{V} \rangle\, (\bar{P}) \vDash \phi[\bar{Q}/\bar{Y}]\end{array}$$

so by the definition of advice lookup:

$$\vdash c\langle \bar{S} \rangle . \ell \langle \bar{V} \rangle \text{ advised by } a\langle \bar{Q} \rangle$$

as required. □

THEOREM (6.5 PARAMETRICITY OF REDUCTION). *Suppose* $\bar{X} \triangleleft \bar{D} \Vdash M : T$ *and* $\vdash \bar{S}$ *and* $\vdash \bar{S} <: \bar{D}[\bar{V}/\bar{X}]$. *Then* $\vdash M[\bar{S}/\bar{X}] \longrightarrow L$ *implies that* $L = N[\bar{S}/\bar{X}]$ *and for all* $\bar{S}'$ *such that* $\vdash \bar{S}' <: \bar{D}[\bar{S}'/\bar{X}]$ *we have that* $\vdash M[\bar{S}'/\bar{X}] \longrightarrow N[\bar{S}'/\bar{X}]$.

Proof. Interesting case is EVAL-LOOKUP, in which case:

$$\begin{array}{l}M \triangleq \texttt{new}\, C(\bar{M}) . \ell \langle \bar{V} \rangle\, (\bar{N}) \\ L \triangleq (\texttt{new}\, C(\bar{M}))[\bar{S}/\bar{X}] . \ell \langle \bar{V}[\bar{S}/\bar{X}] \rangle\, [\bar{A}]\, (\bar{N}[\bar{S}/\bar{X}]) \\ \bar{A} \triangleq \left[a\langle \bar{U} \rangle \;\middle|\; \begin{array}{l}\mathscr{D} \ni \texttt{advice}\, a\ \cdots \\ \vdash C[\bar{S}/\bar{X}] . \ell \langle \bar{V}[\bar{S}/\bar{X}] \rangle \text{ advised by } a\langle \bar{U} \rangle\end{array} \right]\end{array}$$

Since $\bar{X} \triangleleft \bar{D} \Vdash M : T$ we have $\bar{X} \triangleleft \bar{D} \Vdash \mathsf{ok}$, $\bar{X} \triangleleft \bar{D} \vdash C$, so we can use Propositions D.7 (Inverse substitutivity of advice lookup) and D.1 (Substitutivity of advice lookup) to get:

$$\begin{array}{l}\bar{A} \triangleq \bar{B}[\bar{S}/\bar{X}] \\ \bar{B} \triangleq \left[a\langle \bar{Q} \rangle \;\middle|\; \begin{array}{l}\mathscr{D} \ni \texttt{advice}\, a\ \cdots \\ \vdash C . \ell \langle \bar{V} \rangle \text{ advised by } a\langle \bar{Q} \rangle\end{array} \right]\end{array}$$

and so:

$$L \triangleq N[\bar{S}/\bar{X}]$$
$$N \triangleq \texttt{new}\ C(\bar{M}).\ell\langle\bar{V}\rangle\,[\bar{B}]\,(\bar{N})$$

Moreover, for any $\bar{S}'$ such that $\vdash \bar{S}' <: \bar{D}[\bar{S}'/\bar{X}]$ we have:

$$M[\bar{S}'/\bar{X}] \to L'$$
$$L' \triangleq (\texttt{new}\ C(\bar{M}))[\bar{S}/\bar{X}].\ell\langle\bar{V}[\bar{S}/\bar{X}]\rangle\,[\bar{A}']\,(\bar{N}[\bar{S}/\bar{X}])$$
$$\bar{A}' \triangleq \left[a\langle\bar{U}'\rangle \;\middle|\; \begin{array}{l} \mathscr{D} \ni \texttt{advice}\ a\ \cdots \\ \vdash C[\bar{S}'/\bar{X}].\ell\langle\bar{V}[\bar{S}'/\bar{X}]\rangle\ \mathsf{advised\ by}\ a\langle\bar{U}'\rangle \end{array} \right]$$

and again, we can use Propositions D.7 (Inverse substitutivity of advice lookup) and D.1 (Substitutivity of advice lookup) to get:

$$\bar{A}' \triangleq \bar{B}[\bar{S}'/\bar{X}]$$

and hence:

$$L' = N[\bar{S}'/\bar{X}]$$

as required. □