

A Formal Semantics of Flexible and Safe Pointcut/Advice Bindings

Bruno De Fraine¹, Mario Südholt² and Viviane Jonckers¹

¹ System and Software Engineering Lab (SSEL), Vrije Universiteit Brussel

² OBASCO project, EMN-INRIA, LINA

Technical Report SSEL 02/2007/a
Vrije Universiteit Brussel

29 October 2007

Abstract

AspectJ was designed as a seamless aspect-oriented extension of the Java programming language. However, unlike Java, AspectJ does not have a safe type system: an accepted binding between a pointcut and an advice can give rise to type errors at runtime. In addition, AspectJ's typing rules are severely restricting the definition of certain generic advice behavior.

In this technical report, we present a formal semantics and a type system for an extension of AspectJ that addresses these issues. The type system, in particular, ensures safety for both generic and non-generic pointcut/advice declarations. Pointcuts quantify over heterogeneous sets of join points and are hence typed using type ranges in our approach, while type variables and a separate signature for proceed declarations allow to express the generic and invasive nature of advices. Using these mechanisms, we can express advice that augments, narrows or replaces base functionality in possibly generic contexts.

Concretely, this report briefly reviews the type safety problems of pointcut/advice bindings in AspectJ-like languages and previous (partial) solutions to these problems that have been proposed in the literature. We then give a brief overview of our approach. Finally, we give a complete formal account of our approach by presenting evaluation rules, typing rules, type preservation and progress theorems as well as corresponding proofs.

Contents

1	Introduction	2
1.1	Type-safety in AspectJ-like languages	2
1.1.1	Problem statement	2
1.1.2	Previous solutions	3
1.2	Overview of our approach	4
2	(Cast-Free) Featherweight Generic Java	5
2.1	FGJ Syntax	5
2.2	FGJ Dynamics	6
2.2.1	Lookup	6
2.2.2	Evaluation	6
2.3	FGJ Statics	8
2.3.1	Auxiliary Judgments	8
2.3.2	Term Typing	8
2.3.3	Declaration Typing	8
2.4	Properties	9
3	Flexible and Safe Pointcut/Advice Bindings	10
3.1	Syntax	10
3.2	Dynamic Semantics	11
3.2.1	Pointcut Matching	11
3.2.2	Evaluation	11
3.3	Static Semantics	12
3.3.1	Auxiliary Judgments and Term Typing	12
3.3.2	Pointcut and Advice Typing	12
3.4	Properties	15
3.4.1	Preservation	15
3.4.2	Progress	17
4	AspectJ-like pointcuts	18
4.1	Matching and Typing Judgements	18
4.2	Consistency Property	19
5	Conclusion	21
	Bibliography	22

Chapter 1

Introduction

The work we report on here is motivated by (well-known) problems of aspects in AspectJ-like aspect languages concerning the typing of advice and uses of the `proceed` instruction in advice. In this section we first briefly review these problems and previous partial solutions to them. We then give an overview of our approach.

1.1 Type-safety in AspectJ-like languages

The type system that is presented in this technical report has been developed in order to answer different open questions concerning the correct typing of AspectJ-like around advice that call `proceed`, in particular, if generic advice behavior is desirable and in the context the application of advice to generic base functionality.

We now first make this problem statement more concrete by discussing three groups of problems that make explicit the typing problems and corresponding limitations of use that AspectJ-like aspects are subject to. Where appropriate, we discuss these problems based on references to appropriate previous work. In a second step, we then briefly discuss previous approaches that resolved some of these issues.

However, note that this section is not intended to discuss these problems in detail or to give a detailed overview on the relevant related work, because the purposes of this report is to give a complete definition of our formal approach: an account that is more oriented towards language design issues and that includes more examples and discussion of related work is in preparation.

1.1.1 Problem statement

The typing problems of advice and `proceed` becomes manifest in or influence a large number of different contexts in that non-generic AspectJ aspects are applied to non-/generic Java based programs. In the following we briefly review three (groups) of such programming contexts and the corresponding typing problems:

- Unsoundness of the typing of `around` and `proceed`
- Aspects and generic contexts
- Advice with `Object` return type

Unsoundness of the typing of around and proceed. The typing strategy of around advice AspectJ is unsound in general. As noted by Wand et al. [6] for an early (non-generic) version of AspectJ, the type of **proceed** would have to be determined in terms of all joinpoints that can trigger the corresponding advice, which is not reasonably tractable, e.g., in the presence of abstract pointcuts. AspectJ uses the heuristics to assign the return type of the advice also for corresponding **proceed** instruction, which produces a runtime error if the underlying joinpoint expects an incompatible type. Wand et al. show that a valid typing for AspectJ’s **proceed** would not allow any variance in its return type.

Aspects and generic contexts. Jagadeesan et al. [3] extend this problem analysis by a number of deficiencies of recent (generic) versions of AspectJ that provide some support for generics in base programs and aspects. They show that generic aspects that cannot be expressed with AspectJ are useful even for non-generic base classes, that covariant return types pose difficulties in the presence of aspects, that handling of contravariant argument types is problematic and, finally, that (generic and non-generic) advice complicate the type erasure strategy to typing that is commonly used in Java-based languages and systems.

Advice with Object return type. AspectJ employs an ad hoc type variance rule to support a special class of generic advice that is characterized by always invoking the **proceed**-method with an original argument value from the join point caller and always returning a value obtained from the join point through a **proceed** invocation (in which case these values will always be of a correct type).

This ad hoc rule states that when the return type of an *around*-advice is declared as **Object**, the default binding rule does not apply and the advice can instead be combined with any join point return type. Put differently, the typing rules consider the **Object** return type as a necessary and sufficient condition for an advice to be generic with respect to its return value. However, this condition is neither necessary nor sufficient, and this causes type errors on the one hand, and prevents the typing of certain valid advice methods on the other hand.

1.1.2 Previous solutions

Remedies to some of the above-mentioned deficiencies have been presented in two recent articles.

Jagadeesan et al. [3] extend Featherweight Generic Java [2] with an advice construct whose type may depend on explicitly-declared type variables. This enables typing of advice similar to the generic advice proposed in this report. They present two type-safe systems, one based on a type-carrying semantics and another based on type erasure. However, their approach assigns equal signatures to **proceed** and the corresponding advice, and must therefore require the join point signature to be equal to the these signature (i.e. no type variance is admitted, although the type variables from the signatures can be instantiated for each join point). The typing of *replacement* advice is therefore restricted.

Clifton and Leavens [1] introduce an imperative core language that models context-exposing pointcut primitives as well as **around**-advice capable of changing parameter bindings on **proceed**-invocations. They define how argument types and a return type for pointcut expressions can be derived that correspond to the static types of any join point matched by the pointcut. In a binding, the return type of an advice can be a subtype of the return type associated with the pointcut, but **proceed** will always employ the return type of the advised methods. Similar to our approach, advice and **proceed**-signatures can thus be different to allow more liberal bindings while maintaining soundness. However, the approach does not allow similar type variance for the

argument types (relaxation of this restriction is cited as future work). Also, there is no support for generic advice (the languages employs non-variable types only).

1.2 Overview of our approach

Our proposal tackles these problems based on three principles that support a more powerful and flexible type system than previous ones:

- Separate type signatures for advice and **proceed**
- Type ranges and corresponding type variance relations
- Advice type variables

Separate type signatures for advice and proceed. Similar to, e.g., Jagadeesan et al. [3], we allow type variables to be declared at the level of advice to support generic advice behavior in our type system.

Type ranges and corresponding type variance relations. The behavior of an **around**-advice can be seen as being governed by two interfaces: the **proceed**-interface determines what the advice *expects* from the join point, while the advice interface determines what it *provides* to the join point caller. We use an explicit **proceed**-signature in addition to the regular advice signature for **around**-advice in order to account for this distinction. The advice body should adhere to the advice signature as before, but it can only employ a **proceed**-method with the declared **proceed**-signature.

Advice type variables. Variance relations between join points, advice and **proceed** instructions involve the signatures of advice and join points, but we have to enforce them in the typing rules using a pointcut signature, which functions as an abstraction of a set of (possibly heterogeneous) join points. Since the join point signature is bound on both ends by the two signature relations, we propose to type pointcuts with *signature ranges* (rather than a single signature), which in turn consist of *type ranges* (rather than a single type) in the position of arguments and return values. A range is described using a lower bound (the most specific signature/type) and upper bound (most general signature/type).

Chapter 2

(Cast-Free) Featherweight Generic Java

The formalization that we develop in this document is built on top of Featherweight Generic Java (FGJ) [2]. To make the text self-contained, this section will summarize the FGJ language by presenting the definition of its relevant syntax, evaluation rules and typing rules. The reader is referred to the original material for a detailed description of FGJ.

FGJ reduces Java (plus its Generics feature) to a typed functional calculus without mutable state. Field declarations define structured data that can be constructed through constructor calls and deconstructed through field accesses. Method declarations, variables and method calls provide a term substitution mechanism. While classes can inherit field and method declarations, FGJ has no support for interfaces or overloading. Although the original FGJ definition from [2] included casts to investigate type erasure, we employ a cast-free language in the same way as [3], since, as argued there, casting issues are believed to be orthogonal to problems of join point and advice compatibility.

2.1 FGJ Syntax

The definition of the FGJ syntax is presented in figure 2.1. This syntax definition uses some notational conventions from the original presentation of the Featherweight Java calculus in [2]. In particular, we indicate the syntactic structure of an expression by means of metavariables, which are the non-terminals in the grammar, along with the lexical metavariables shown at the top of the figure. Additionally, we write $\bar{e}$ for an ordered sequence of zero or more elements $e_1, \dots, e_n$, where the element separator may be space, comma or semicolon, depending on the context. This convention is sometimes extended across binary constructs where the elements of two sequences should be appropriately ‘zipped’, e.g. $\bar{T} \bar{f}$ signifies $T_1 f_1, \dots, T_n f_n$. An individual element is referred to as e_i . When no type variables or type arguments are given, the surrounding angle brackets may be omitted as a form of syntactic sugar.

The syntax follows the Java original with the following exceptions. From [2], we adapt the usage of “ $\triangleleft$ ” as an abbreviation for “**extends**”, and of “ $\bullet$ ” to signify the empty type or term environment. As in [3], we omit the “**return**” keyword and trailing semicolon from the body of a method declaration. Identical to [4], we omit constructors from class definitions since they follow directly from the field definitions in FGJ and thus convey no additional information.

c, d	Class names (including Object)
f, g, h	Field names
$\ell, \bar{h}$	Method names
x, y, z	Term variables (including this)
X, Y, Z, W	Type variables
$C, D, E, F, G ::= c\langle\bar{T}\rangle$	Non-variable types
$P, Q, R, S, T, U, V ::= X \mid C$	Types
$M, N, I, J, K, L ::=$	Terms
x	Term variable
$M.f$	Field access
$M.\ell\langle\bar{T}\rangle(\bar{N})$	Method call
$\text{new } C(\bar{N})$	Constructor call
$\mu ::= \langle\bar{X} \triangleleft \bar{C}\rangle R \ell(\bar{P} \bar{x})\{M\}$	Method declarations
$\mathcal{D} ::=$	Top level declarations
$\text{class } c\langle\bar{X} \triangleleft \bar{C}\rangle \triangleleft D\{\bar{T} \bar{f}; \bar{\mu}\}$	Class declarations
$\Gamma ::= \bullet \mid \Gamma, T \ x$	Term environment
$\Delta ::= \bullet \mid \Delta, X \triangleleft C$	Type environment

Figure 2.1: FGJ Syntax

2.2 FGJ Dynamics

All of the following definitions are with respect to a fixed set of declarations (represented by $\mathcal{D}$, as in [3]). FGJ assumes that **Object** is not declared in this set, and that its induced subtype relation contains no cycles [2].

2.2.1 Lookup

Field and method lookup are represented by the auxiliary functions $\text{fields}(T)$ and $\text{meth}(T.\ell)$, defined in figure 2.2. For each function, there are two rules for nonvariable types; they recursively follow the parent link up to **Object** (which is empty in FGJ). Each function also has a rule for variable types, where the variable's bound from the type environment is considered. When doing lookup for non-variable types, the type environment is not considered, and may be omitted. This is the case in the premises of these rules; it will also be the case during evaluation, when the type environment is empty.

2.2.2 Evaluation

The evaluation rules are given in figure 2.3. The evaluation of a field access and a method call is straightforward using the auxiliary lookup functions from the previous section. Finally, we can have congruent evaluation in the context of a target (of field access and method call) or argument (of method or constructor call).

$$\begin{array}{c}
\text{FIELD-OBJECT} \\
\hline
\Delta \vdash \text{fields}(\text{Object}) = \bullet
\end{array}
\quad
\begin{array}{c}
\text{FIELD-THIS-SUPER} \\
\mathcal{D} \ni \text{class } c \langle \bar{X} \rangle \triangleleft D \{ \bar{T} \ \bar{f}; \dots \} \\
\vdash \text{fields}(D[\bar{V}/\bar{X}]) = \bar{S} \ \bar{g} \\
\hline
\Delta \vdash \text{fields}(c \langle \bar{V} \rangle) = \bar{S} \ \bar{g}, \bar{T}[\bar{V}/\bar{X}] \ \bar{f}
\end{array}
\quad
\begin{array}{c}
\text{FIELD-VAR} \\
\vdash \text{fields}(\Delta(X)) = \bar{T} \ \bar{f} \\
\hline
\Delta \vdash \text{fields}(X) = \bar{T} \ \bar{f}
\end{array}$$

$$\begin{array}{c}
\text{METHOD-THIS} \\
\mathcal{D} \ni \text{class } c \langle \bar{X} \rangle \dots \{ \dots \langle \bar{Y} \rangle \triangleleft \bar{E} \rangle R(\bar{P} \ \bar{x}) \{ M \} \} \\
\hline
\Delta \vdash \text{meth}(c \langle \bar{V} \rangle . \ell) = \langle \bar{Y} \rangle \triangleleft \bar{E} \rangle R(\bar{P} \ \bar{x}) \{ M \} [\bar{V}/\bar{X}]
\end{array}$$

$$\begin{array}{c}
\text{METHOD-SUPER} \\
\mathcal{D} \ni \text{class } c \langle \bar{X} \rangle \triangleleft D \{ \dots \bar{\mu} \} \quad \ell \text{ not defined in } \bar{\mu} \\
\vdash \text{meth}(D[\bar{V}/\bar{X}].\ell) = \langle \bar{Y} \rangle \triangleleft \bar{E} \rangle R(\bar{P} \ \bar{x}) \{ M \} \\
\hline
\Delta \vdash \text{meth}(c \langle \bar{V} \rangle . \ell) = \langle \bar{Y} \rangle \triangleleft \bar{E} \rangle R(\bar{P} \ \bar{x}) \{ M \}
\end{array}
\quad
\begin{array}{c}
\text{METHOD-VAR} \\
\vdash \text{meth}(\Delta(X). \ell) = \langle \bar{Y} \rangle \triangleleft \bar{E} \rangle R(\bar{P} \ \bar{x}) \{ M \} \\
\hline
\Delta \vdash \text{meth}(X. \ell) = \langle \bar{Y} \rangle \triangleleft \bar{E} \rangle R(\bar{P} \ \bar{x}) \{ M \}
\end{array}$$

Figure 2.2: Field ($\Delta \vdash \text{fields}(T) = \bar{S} \ \bar{f}$) and method ($\Delta \vdash \text{meth}(T. \ell) = \langle \bar{Y} \rangle \triangleleft \bar{E} \rangle R(\bar{P} \ \bar{x}) \{ M \}$) lookup

$$\begin{array}{c}
\text{EVAL-FIELD} \\
\vdash \text{fields}(C) = \bar{f} \\
\hline
\text{new } C(\bar{N}).f_i \rightarrow N_i
\end{array}
\quad
\begin{array}{c}
\text{EVAL-METHOD} \\
\vdash \text{meth}(C. \ell) = \langle \bar{X} \rangle (\bar{x}) \{ L \} \quad M = \text{new } C(\dots) \\
\hline
M. \ell \langle \bar{V} \rangle (\bar{N}) \rightarrow L[\bar{V}/\bar{X}, M/\text{this}, \bar{N}/\bar{x}]
\end{array}$$

$$\begin{array}{c}
\text{EVAL-CONTEXT} \\
L \rightarrow L' \\
\hline
\begin{array}{l}
L. \ell \langle \bar{V} \rangle (\bar{N}) \rightarrow L'. \ell \langle \bar{V} \rangle (\bar{N}) \quad M. \ell \langle \bar{V} \rangle (\dots, L, \dots) \rightarrow M. \ell \langle \bar{V} \rangle (\dots, L', \dots) \\
L.f \rightarrow L'.f \quad \text{new } C(\dots, L, \dots) \rightarrow \text{new } C(\dots, L', \dots)
\end{array}
\end{array}$$

Figure 2.3: FGJ Evaluation ($M \rightarrow M'$)

$$\begin{array}{c}
\text{SUB-VAR-EXTENDS} \\
\Delta(X) = \triangleleft C \\
\hline
\Delta \vdash X <: \bar{C}
\end{array}
\quad
\begin{array}{c}
\text{SUB-CLASS} \\
\mathcal{D} \ni \text{class } c \langle \bar{X} \rangle \triangleleft D \dots \\
\hline
\Delta \vdash c \langle \bar{V} \rangle <: D[\bar{V}/\bar{X}]
\end{array}
\quad
\begin{array}{c}
\text{SUB-REFLEX} \\
\hline
\Delta \vdash T <: T
\end{array}
\quad
\begin{array}{c}
\text{SUB-TRANS} \\
\Delta \vdash T <: T' \\
\Delta \vdash T' <: T'' \\
\hline
\Delta \vdash T <: T''
\end{array}$$

$$\begin{array}{c}
\text{TYPE-VAR} \\
\Delta(X) \text{ defined} \\
\hline
\Delta \vdash X
\end{array}
\quad
\begin{array}{c}
\text{TYPE-CLASS} \\
\mathcal{D} \ni \text{class } c \langle \bar{X} \rangle \triangleleft \bar{C} \dots \quad \Delta \vdash \bar{V} \\
\hline
\Delta \vdash c \langle \bar{V} \rangle
\end{array}
\quad
\begin{array}{c}
\Delta \vdash \bar{V} <: \bar{C}[\bar{V}/\bar{X}]
\end{array}
\quad
\begin{array}{c}
\text{TYPE-OBJECT} \\
\hline
\Delta \vdash \text{Object}
\end{array}$$

$$\begin{array}{c}
\text{ENV-TYPE} \\
\forall i. X_1 \triangleleft C_1, \dots, X_n \triangleleft C_n \vdash C_i \\
\forall i, j. X_i = X_j \text{ implies } i = j \\
\hline
X_1 \triangleleft C_1, \dots, X_n \triangleleft C_n \vdash \text{ok}
\end{array}
\quad
\begin{array}{c}
\text{ENV-EMPTY} \\
\Delta \vdash \text{ok} \\
\hline
\Delta; \bullet \vdash \text{ok}
\end{array}
\quad
\begin{array}{c}
\text{ENV-TERM-VAR} \\
\Delta; \Gamma \vdash \text{ok} \quad \Delta \vdash T \\
\Gamma(x) \text{ undefined} \\
\hline
\Delta; \Gamma, T \ x \vdash \text{ok}
\end{array}$$

Figure 2.4: FGJ Subtyping ($\Delta \vdash T <: T'$), well-formed types ($\Delta \vdash T$) and well-formed environments ($\Delta \vdash \text{ok}$) ($\Delta; \Gamma \vdash \text{ok}$)

$$\begin{array}{c}
\text{TERM-VAR} \\
\frac{\Delta; \Gamma \vdash \text{ok} \quad \Gamma(x) = T}{\Delta; \Gamma \vdash x : T}
\end{array}
\qquad
\begin{array}{c}
\text{TERM-OBJECT} \\
\frac{\Delta; \Gamma \vdash \text{ok} \quad \vdash \text{fields}(C) = \bar{S} \bar{f} \quad \Delta \vdash C \quad \Delta; \Gamma \vdash \bar{N} : \bar{S}' \quad \Delta \vdash \bar{S}' <: \bar{S}}{\Delta; \Gamma \vdash \text{new } C(\bar{N}) : C}
\end{array}$$

$$\begin{array}{c}
\text{TERM-FIELD} \\
\frac{\Delta \vdash \text{fields}(T) = \bar{S} \bar{f} \quad \Delta; \Gamma \vdash M : T}{\Delta; \Gamma \vdash M.f_i : S_i}
\end{array}
\qquad
\begin{array}{c}
\text{TERM-METHOD} \\
\frac{\Delta \vdash \text{meth}(T.\ell) = \langle \bar{Y} \triangleleft \bar{E} \rangle R(\bar{P}) \dots \quad \Delta \vdash \bar{V} \quad \Delta; \Gamma \vdash (M, \bar{N}) : (T, \bar{P}') \quad \Delta \vdash (\bar{V}, \bar{P}') <: (\bar{E}, \bar{P})[\bar{V}/\bar{Y}]}{\Delta; \Gamma \vdash M.\ell \langle \bar{V} \rangle (\bar{N}) : R[\bar{V}/\bar{Y}]}
\end{array}$$

Figure 2.5: FGJ Term Typing ($\Delta; \Gamma \vdash M : T$)

2.3 FGJ Statics

2.3.1 Auxiliary Judgments

The definitions of subtyping, well-formed types and well-formed environments are given in figure 2.4.

A variable type is a subtype of its bound in the environment, while a non-variable type is a subtype of the parent given in the class declaration (after substituting type arguments for the type variables). The subtype relation is also reflexive and transitive. A variable type is well-formed if it is defined in the type environment.

A non-variable type is well-formed if its type arguments are well-formed, and if they lie below the declared bounds of the respective type variables (again after substituting type arguments for the type variables). The built-in `Object` type is of course also well-formed.

Finally, the bounds of well-formed type environment have to be well-formed in that environment, and the environment may define each type variable at most once. When a type environment is extended with a term environment, the combination is well-formed if additionally all the term types are well-formed in the type environment, and if each term variable is defined at most once.

2.3.2 Term Typing

The rules for typing the different term forms are given in figure 2.5. The requirement that typing can only occur with a well-formed environment is realized by including this condition in the cases of a term variable and a constructor call; in the other cases this is enforced inductively since the receiver is required to be well-typed in the same environment. The typing rules also employ method and field lookup, and roughly require that the variable, field and method names are defined, that all given types are well-formed, and that the given arguments belong to the declared argument or field types. The declared return type, field type or variable type is also used as the result of the typing.

2.3.3 Declaration Typing

Figure 2.6 present the rules for well-formed overriding and well-formed method and class declarations. Method overriding is required to be invariant in parameter types and type variable bounds, and covariant in return type. Beside a valid overriding, method declaration typing requires well-formed types and a method body term that has suitable type in the environment

$\frac{\text{OVERRIDE-UNDEFINED} \quad \vdash \text{meth}(D.\ell) \text{ undefined}}{\vdash \langle \bar{Y} \triangleleft \bar{E} \rangle R(\bar{P}) \text{ can override } D.\ell}$	$\frac{\text{OVERRIDE-DEFINED} \quad \vdash \text{meth}(D.\ell) = \langle \bar{Y} \triangleleft \bar{E} \rangle R(\bar{P}) \dots \quad \bar{Y} \triangleleft \bar{E} \vdash R' <: R}{\vdash \langle \bar{Y} \triangleleft \bar{E} \rangle R'(\bar{P}) \text{ can override } D.\ell}$
$\frac{\text{DEC-METHOD} \quad \begin{array}{l} \bar{X} \triangleleft \bar{C}, \bar{Y} \triangleleft \bar{E} \vdash \bar{E}, \bar{P}, R \\ \bar{X} \triangleleft \bar{C}, \bar{Y} \triangleleft \bar{E}; \bar{P} \ \bar{x}, c \langle \bar{X} \rangle \text{ this } \vdash M : R' \\ \bar{X} \triangleleft \bar{C}, \bar{Y} \triangleleft \bar{E} \vdash R' <: R \\ \vdash \langle \bar{Y} \triangleleft \bar{E} \rangle R(\bar{P} \ \bar{x}) \text{ can override } D.\ell \end{array}}{\vdash \langle \bar{Y} \triangleleft \bar{E} \rangle R \ \ell(\bar{P} \ \bar{x}) \{M\} : \text{ok in } c \langle \bar{X} \triangleleft \bar{C} \rangle \triangleleft D}$	$\frac{\text{DEC-CLASS} \quad \begin{array}{l} \bar{X} \triangleleft \bar{C} \vdash \bar{C}, D, \bar{T} \\ \vdash \text{fields}(D) = \bar{S} \ \bar{g} \\ \bar{X} \triangleleft \bar{C}; \bar{S} \ \bar{g}, \bar{T} \ f \vdash \text{ok} \\ \vdash \bar{\mu} : \text{ok in } c \langle \bar{X} \triangleleft \bar{C} \rangle \triangleleft D \end{array}}{\vdash \text{class } c \langle \bar{X} \triangleleft \bar{C} \rangle \triangleleft D \{ \bar{T} \ \bar{f}; \bar{\mu} \}}$

Figure 2.6: Well-formed Overriding ($\vdash \langle \bar{Y} \triangleleft \bar{E} \rangle R(\bar{P}) \text{ can override } D.\ell$), Method Declaration Typing ($\vdash \mu : \text{ok in } c \langle \bar{X} \triangleleft \bar{C} \rangle \triangleleft D$) and Class Declaration Typing ($\vdash \text{class } c \langle \bar{X} \triangleleft \bar{C} \rangle \triangleleft D \{ \bar{T} \ \bar{f}; \bar{\mu} \}$)

that follows from the declaration. Note that since this term typing requires a well-formed environment, unique term and type variables are enforced. Similarly, well-formed types and a well-formed environment are required for class declarations, to prevent duplicate field names (which would otherwise render the field lookup undeterministic).

2.4 Properties

The dynamic and static semantics defined in the previous sections are then consistent: as shown in [2], the FGJ language has the classic type preservation and progress properties under the assumption that all top-level declarations are well-typed (i.e. $\vdash \mathcal{D}_i$ for each $\mathcal{D}_i$ in $\mathcal{D}$). We repeat the formal definition of these properties below.

Theorem 2.1 (Preservation). *If $\vdash M : T$ and $M \rightarrow M'$ then $\vdash M' : T'$ for some T' such that $\vdash T' <: T$.*

The progress property uses the notion of a value. A value is a special kind of term, which is formally defined through the following recursive syntax specification:

$$H ::= \text{new } C(\bar{H})$$

Theorem 2.2 (Progress). *If $\vdash M : T$ then either M is a value or $M \rightarrow M'$ holds for some M' .*

Chapter 3

Flexible and Safe Pointcut/Advice Bindings

3.1 Syntax

The syntax modifications of our approach are shown in figure 3.1. A top level advice declaration is provided; it includes a declaration of type variables and a dual advice signature. The advice declaration also includes a pointcut expression, but we will abstract over its concrete form. As such, we aim to define a framework that is agnostic of a particular pointcut language. (An integration of AspectJ-like pointcut expressions is discussed in chapter 4 though.)

Since the advice body may invoke the original join point, the proceed call is included in the terms, and the term environment can include a proceed signature. Advised method calls are included as another term form; while they are not allowed as such in the language, we need them to describe the process of advice application in running code.

Finally, we include the possibility for type environments to specify a lower bound for a type variable in addition to (or in place of) an upper bound. The notation “ $\triangleright$ ” is used here as an abbreviation for “**super**”, the complement of “**extends**”.

a, b	Advice names
$M, N, I, J, K, L ::= \dots$	Terms
$M.\ell\langle T \rangle[\bar{a}](\bar{N})$	Advised method call
proceed ($\bar{N}$)	Proceed call
$\phi ::= \dots$	Pointcuts (abstract)
$\mathcal{D} ::= \dots$	Top level declarations
advice $a\langle \bar{X} \triangleleft \bar{C} \rangle R(\bar{P} \bar{x}) : \phi : S(\bar{Q}) \{M\}$	Advice declarations
$\Gamma ::= \dots \mid \Gamma, S \text{ proceed}(\bar{Q})$	Term environment
$\Delta ::= \dots \mid \Delta, X \triangleright C$	Type environment

Figure 3.1: Syntax modifications

$$\begin{array}{c}
\text{EVAL-SELECT} \\
\frac{\bar{b} = [a | \mathcal{D} \ni \text{advice } a \cdots : \cdots : \cdots]}{M.\ell < \bar{V} > (\bar{N}) \rightarrow M.\ell < \bar{V} > [\bar{b}] (\bar{N})}
\end{array}
\quad
\begin{array}{c}
\text{EVAL-METHOD} \\
\frac{\vdash \text{meth}(C.\ell) = < \bar{X} > (\bar{x}) \{L\} \quad M = \text{new } C(\cdots)}{M.\ell < \bar{V} > [] (\bar{N}) \rightarrow L[\bar{V}/\bar{x}, M/\text{this}, \bar{N}/\bar{x}]}
\end{array}$$

$$\begin{array}{c}
\text{EVAL-APPLY} \\
\frac{\mathcal{D} \ni \text{advice } a(\bar{x}) : \phi : \cdots \{L\} \quad \vdash \text{match}(M.\ell < \bar{V} > (\bar{N}), \phi) = \bar{y}; \bar{K}; I.\bar{h} < \bar{U} > (\bar{J})}{M.\ell < \bar{V} > [a, \bar{b}] (\bar{N}) \rightarrow L[\bar{x}[\bar{K}/\bar{y}]/\bar{x}, I.\bar{h} < \bar{U} > [\bar{b}] (\bar{J}) / \text{proceed}(\bar{x})][\bar{K}/\bar{y}]}
\end{array}$$

$$\begin{array}{c}
\text{EVAL-NOAPPLY} \\
\frac{\mathcal{D} \ni \text{advice } a \cdots : \phi : \cdots \quad \vdash \text{match}(M.\ell < \bar{V} > (\bar{N}), \phi) = \perp}{M.\ell < \bar{V} > [a, \bar{b}] (\bar{N}) \rightarrow M.\ell < \bar{V} > [\bar{b}] (\bar{N})}
\end{array}$$

Figure 3.2: Method call evaluation rules

3.2 Dynamic Semantics

3.2.1 Pointcut Matching

Although pointcuts are left abstract, we assume that some definition of their matching exists in the form of the `match` function with general form:

$$\vdash \text{match}(M.\ell < \bar{V} > (\bar{N}), \phi) = \bar{y}; \bar{K}; I.\bar{h} < \bar{U} > (\bar{J})$$

For a given method call $(M.\ell < \bar{V} > (\bar{N}))$ and pointcut (ϕ) , if it matches, the function gives the exposed variables $(\bar{y})$, their values $(\bar{K})$ and a proceed body $(I.\bar{h} < \bar{U} > (\bar{J}))$ parametrized with those variables. If there is no match, the value $\perp$ is returned.

We require that for any method call with sufficiently evaluated arguments, a pointcut will either match it or not match it.

The aspect programmer is likely to expect that employing `proceed` with the original arguments will return the original join point, i.e. the following condition:

$$(I.\bar{h} < \bar{U} > (\bar{J}))[\bar{K}/\bar{y}] = M.\ell < \bar{V} > (\bar{N})$$

While this relation could be regarded as a valuable design principle for a pointcut language, it is not required by the framework.

3.2.2 Evaluation

The evaluation rules EVAL-FIELD and EVAL-CONTEXT are adapted from FGJ without modification. The evaluation of a method call, however, is altered to accommodate for possible advice applications, as shown in figure 3.2.

The substitutions are required to match $\bar{x}$ to $\bar{y}$. (Note that typing will enforce that $\bar{x}$ are all different, $\bar{y}$ are all different, and $\bar{x} \subseteq \bar{y}$.) Proceed substitution is defined as homomorphic for all terms constructs except `proceed`, where:

$$\text{proceed}(\bar{N})[M/\text{proceed}(\bar{x})] = M[\bar{N}/\bar{x}]$$

$$\begin{array}{c}
\text{SUB-VAR-SUPER} \\
\frac{\Delta(X) = \triangleright C}{\Delta \vdash C <: \bar{X}} \\
\\
\text{TERM-ADVISED} \\
\frac{\Delta; \Gamma \vdash M.\ell <\bar{V}>(\bar{N}) : R \quad \mathcal{D} \ni \text{advice } \bar{a} \cdots}{\Delta; \Gamma \vdash M.\ell <\bar{V}>[\bar{a}](\bar{N}) : R} \\
\\
\text{ENV-PROCEED} \\
\frac{\Delta; \Gamma \vdash \text{ok} \quad \Delta \vdash \bar{Q}, S \quad \Gamma(\text{proceed}) \text{ undefined}}{\Delta; \Gamma, S \text{ proceed}(\bar{Q}) \vdash \text{ok}} \\
\\
\text{TERM-PROCEED} \\
\frac{\Gamma; \Delta \vdash \text{ok} \quad \Gamma(\text{proceed}) = S(\bar{Q}) \quad \Delta; \Gamma \vdash \bar{N} : \bar{Q}' \quad \Delta \vdash \bar{Q}' <: \bar{Q}}{\Delta; \Gamma \vdash \text{proceed}(\bar{N}) : S}
\end{array}$$

Figure 3.3: Additional rules for subtyping, well-formed environments and term typing

$$\begin{array}{c}
\text{WELL-FORMEDNESS-PC} \\
\frac{\vdash \phi : D-E(\bar{F}-\bar{G} \bar{z})}{\vdash D, E, \bar{F}, \bar{G} \quad \forall i, j. z_i = z_j \text{ implies } i = j} \\
\\
\text{CONSISTENCY-PC} \\
\frac{\vdash \text{match}(M, \phi) = \bar{y}; \bar{K}; M' \quad \vdash \phi : D-E(\bar{F}-\bar{G} \bar{z}) \quad \vdash M : R}{\bar{y} = \bar{z} \quad \vdash \bar{K} : \bar{G}' \quad \vdash \bar{G}' <: \bar{G} \quad \bullet; \bar{F} \bar{y} \vdash M' : R' \quad \vdash R' <: E \quad \vdash D <: R}
\end{array}$$

Figure 3.4: Assumptions regarding pointcut typing ($\vdash \phi : D-E(\bar{F}-\bar{G} \bar{z})$)

3.3 Static Semantics

3.3.1 Auxiliary Judgments and Term Typing

The definitions for subtyping, well-formed environments and term typing are extended with new rules to accommodate the new syntax forms, as shown in figure 3.3.

While the subtype relation is supplemented with a new rule (SUB-VAR-SUPER) to interpret the possible lower bound given in type environment, notice that we do not modify the definition of well-formed type environments. Since we only employ the lower bounds when considering subtype relations, they remain excluded from well-formed type environments (and consequently well-typed terms).

The definitions of well-formed environments and term typing are each extended with a straightforward rule to accommodate proceed calls. Another rule is added for the typing of advised method calls: it defers to the typing of the unadvised call and additionally requires that the involved aspects are defined.

3.3.2 Pointcut and Advice Typing

Pointcut Typing

Although we abstract over the concrete form of pointcuts in our framework, we consider a typing judgement for pointcuts with the general form:

$$\vdash \phi : D-E(\bar{F}-\bar{G} \bar{z})$$

For a pointcut (ϕ), this judgment describes the variables it exposes ($\bar{z}$) as well as the type ranges for these variable values ($\bar{F}-\bar{G}$) and the return value ($D-E$). Figure 3.4 presents our

DEC-ADVICE

$$\begin{array}{c}
\bar{X} \triangleleft \bar{C} \vdash \bar{C}, R, \bar{P}, S, \bar{Q} \\
\bar{X} \triangleleft \bar{C}; \bar{P} \bar{x}, S \text{ proceed}(\bar{Q}) \vdash M : R' \\
\bar{X} \triangleleft \bar{C} \vdash R' <: R \\
\vdash \phi : D-E(\bar{F}-\bar{G} \bar{z}) \\
\hline
\bar{Z}, W \text{ fresh} \quad \exists \bar{V}. \left(\begin{array}{cc} \bar{X} \triangleleft \bar{C}, \bar{Z} \triangleleft \bar{G}, W \triangleleft E \vdash \bar{V}, \bar{V} <: \bar{C} & \bar{Z} \triangleright \bar{F} \vdash \bar{Q}[\bar{V}/\bar{X}] <: \bar{x}[\bar{Z}/\bar{z}] \\ \bar{Z} \triangleleft \bar{G} \vdash \bar{x}[\bar{Z}/\bar{z}] <: \bar{P}[\bar{V}/\bar{X}] & W \triangleright D \vdash R[\bar{V}/\bar{X}] <: W \end{array} \right) \\
\hline
\vdash \text{advice } a < \bar{X} \triangleleft \bar{C} > R(\bar{P} \bar{x}) : \phi : S(\bar{Q})\{M\}
\end{array}$$

Figure 3.5: Advice typing ($\vdash \text{advice } a < \bar{X} \triangleleft \bar{C} > R(\bar{P} \bar{x}) : \phi : S(\bar{Q})\{M\}$)

assumptions regarding this typing. The assumption WELL-FORMEDNESS-PC specifies that a pointcut can only be typed with well-formed types and with variables that constitute a well-formed environment. We also require the typing of a pointcut to be consistent with its matching semantics, as defined in assumption CONSISTENCY-PC. For any join point M matched by a typed pointcut, we require that the matching binds the variables from the typing (first conclusion) with values that lie below the argument upper bounds from the pointcut typing (second and third conclusion). We require that the proceed term M' can be typed with a term environment that assigns the pointcut lower bounds to the arguments (fourth conclusion), resulting in a type that lies below the upper bound of the return value (fifth conclusion), while the original join point return type should lie above the lower bound (sixth conclusion).

An informal and more high-level interpretation of the assumption CONSISTENCY-PC, is that the proceed term of a match should satisfy $E(\bar{F} \bar{z})$, the upper bound of the pointcut signature range, while $D(\bar{G} \bar{z})$, the lower bound of the signature range, should be stronger than the join point's signature (i.e. it should accept the arguments from the join point caller, and its return value should be accepted by that caller).

Advice Typing

The rule to type an advice declaration is given in figure 3.5. The first three conditions are similar to those of the rule DEC-METHOD from FGJ, except that the declaration of the proceed signature is taken in to account. The fourth condition requires a well-typed pointcut, and the last (grouped) condition relates the pointcut type to the advice and proceed signatures.

This condition employs fresh type variables to represent any possible join point return type (W) and any join point argument types ($\bar{Z}$)¹. Under the assumptions that these join point types adhere to their pointcut bounds (respectively $D-E$ for W and $\bar{F}-\bar{G}$ for $\bar{Z}$), it states that there should exist a binding $\bar{V}$ for the type variables $\bar{X}$ (where $\bar{V}$ can freely employ $W, \bar{Z}$), such that, (i) the binding of the type variables is valid (i.e. the type arguments are well-formed and respect the type variables' bounds) and (ii) the join point types $W, \bar{Z}$ satisfy the type relations with the advice signatures (i.e. $R(\bar{P})$ and $S(\bar{Q})$, after substituting $\bar{V}$ for $\bar{X}$). To line up the types $\bar{Z}$ with $\bar{P}$ and $\bar{Q}$, we employ a substitution to reorder $\bar{Z}$ according to how the elements of $\bar{z}$ are employed in $\bar{x}$. This also enforces that every x_i in the advice declaration belongs to the variables $\bar{z}$ from the pointcut: if not, $\bar{x}[\bar{Z}/\bar{z}]$ will not result in a type in position i and the relation can thus

¹It could be said that these type variables “capture” the join point types similar to how the mechanism of *wildcard capture* [5, 4] allows to represent a wildcard type argument by a fresh type variable.

not be satisfied.

Example Consider the following advice declaration and pointcut type:

$$\begin{aligned} &\text{advice } ex \triangleleft X_1 \triangleleft \text{Person}, X_2 \triangleleft \text{Number} \triangleright X_2(X_1 \ x, \text{Person } y) : \phi : X_2(X_1, \text{Employee}) \cdots \\ &\vdash \phi : \text{Integer-Number}(\text{Person-Person } x, \text{Employee-Person } y) \end{aligned}$$

The pointcut defines two arguments, so consider fresh type variables Z_1, Z_2 for the argument types and fresh variable W for the return type. The advice can be typed with $\bar{V} = Z_1, W$ as a binding for $\bar{X} = X_1, X_2$. Indeed, with this binding we can straightforwardly establish the required conditions: (for brevity, only the relevant parts of the type environments are shown)

$$\begin{aligned} &Z_1 \triangleleft \text{Person}, W \triangleleft \text{Number} \vdash Z_1, W, Z_1 <: \text{Person}, W <: \text{Number} \\ &Z_2 \triangleleft \text{Person} \vdash (Z_1, Z_2) <: (Z_1, \text{Person}) \quad Z_2 \triangleright \text{Employee} \vdash (Z_1, \text{Employee}) <: (Z_1, Z_2) \\ &\vdash W <: W <: W \end{aligned}$$

Contrarily, the following alternative advice cannot be typed with the above pointcut ϕ :

$$\text{advice } ex \triangleleft X_1 \triangleleft \text{Person}, X_2 \triangleleft \text{Number} \triangleright X_2(X_1 \ x, \text{Person } y) : \phi : X_2(\text{Employee}, X_1) \cdots$$

In general, $\bar{V} = V_1, V_2$. We have to establish (amongst other relations) that $Z_1 \triangleleft \text{Person} \vdash Z_1 <: V_1$ and $Z_2 \triangleright \text{Employee} \vdash V_1 <: Z_2$. However, no possible choice for V_1 satisfies both of these relations: Z_1 and Person only satisfy the first, and Z_2 and Employee only satisfy the second.

Side-Track Alternatively (and in other versions of the text), the last condition of the rule DEC-ADVICE was expressed by delegating on the kind (variable or non-variable) of types $\bar{P}$. For example, for each argument x_i , the condition stipulates that following must hold for its types:

$\frac{\text{COND-NONVAR} \quad P_i \neq X_k}{\begin{array}{cc} Q_i \neq X_l & x_i = z_j \\ \vdash G_j <: P_i & \vdash Q_i <: F_j \end{array}}$	$\frac{\text{COND-VAR} \quad P_i = X_k}{\begin{array}{cc} Q_i = X_k & x_i = z_j \\ \forall i'. P_{i'} = X_k \vee Q_{i'} = X_k \text{ implies } i = i' & \vdash G_j <: C_k \end{array}}$
---	---

However, this formulation gives an ad hoc impression, and we can indeed show that our new formulation of the condition is more general. (Again we do this for argument types only, the return type can be easily integrated into this reasoning.)

Proof. For fresh $\bar{Z}, W$, we will show how we can construct a $\bar{V}$ to satisfy the new condition. To determine V_k , we consider X_k : either there is no $P_i = X_k$, in which case we will select $V_k = C_k$, or, there is exactly one $P_i = X_k$ (indeed we required in COND-VAR that this P_i is unique), in which case we will select $V_k = Z_j$ where j is such that $x_i = z_j$.

1. Now, for $\Delta = \{\bar{X} \triangleleft \bar{C}, \bar{Z} \triangleleft \bar{G}\}$, it is clear that $\Delta \vdash \bar{V}$ since $\bar{V}$ only consists of elements of $\bar{C}$ and $\bar{Z}$.
2. It also holds that $\Delta \vdash \bar{V} <: \bar{C}$. Indeed, for each k , either $V_k = C_k$, in which case the relation directly holds, or $V_k = Z_j$, in which case we also know that $P_i = X_k$ with $x_i = z_j$ and thus, per COND-VAR, $\vdash G_j <: C_k$, which entails that $Z_j \triangleleft G_j \vdash V_k = Z_j <: C_k$.

3. The other subtype relations must also hold. For each P_i and corresponding Q_i there are two cases.
 - (a) Either $P_i \neq X_k$, and rule COND-NONVAR guarantees $\vdash G_j <: P_i$ and $\vdash Q_i <: F_j$ for j such that $x_i = z_j$. Consequently, it holds that $Z_j \triangleleft G_j \vdash Z_j <: P_i$ and $Z_j \triangleright F_j \vdash Q_i <: Z_j$, i.e. the required conditions.
 - (b) Otherwise $P_i = Q_i = X_k$, but since we have selected $V_k = Z_j$ for j such that $x_i = z_j$, the required condition becomes $Z_j <: Z_j <: Z_j$ after substitution. This holds for any type environment.

□

3.4 Properties

Similar to the theorems from FGJ, we will proof type safety by showing preservation (subject-reduction) and progress.

3.4.1 Preservation

Lemma 3.1 (Substitutivity). *For $\Delta = \bar{Y} \triangleleft \bar{C}$ and $\bar{D}$ such that $\vdash \bar{D}$ and $\vdash \bar{D} <: \bar{C}[\bar{D}/\bar{Y}]$, it holds that:*

1. *If $\Delta \vdash T$ then $\vdash T[\bar{D}/\bar{Y}]$.*
2. *If $\Delta \vdash T <: S$ then $\vdash T[\bar{D}/\bar{Y}] <: S[\bar{D}/\bar{Y}]$.*

For an additional $\Gamma = \bar{P} \bar{x}$ and $\bar{N}$ such that $\vdash \bar{N} : \bar{Q}$ with $\Delta \vdash \bar{Q} <: \bar{P}$, it holds that:

3. *If $\Delta; \Gamma \vdash M : T$ then $\vdash M[\bar{D}/\bar{Y}, \bar{N}/\bar{x}] : T'$ with $\vdash T' <: T[\bar{D}/\bar{Y}]$.*

And for an additional $\Gamma' = \Gamma, S$ proceed($\bar{Q}$) and M such that $\bullet; \bar{Q} \bar{z} \vdash M : S'$ with $\vdash S' <: S$, it holds that:

4. *If $\Delta; \Gamma' \vdash L : T$ then $\vdash L[\bar{D}/\bar{Y}, \bar{N}/\bar{x}, M/\text{proceed}(\bar{z})] : T'$ with $\vdash T' <: T[\bar{D}/\bar{Y}]$.*

Proof. 1. By induction on T . If T is non-variable and T does not contain $\bar{Y}$, $T[\bar{D}/\bar{Y}] = T$. If T is non-variable and contains $c\langle \bar{Y} \rangle$, then $\vdash T[\bar{D}/\bar{Y}]$ because of rule TYPE-CLASS. Finally, if T is variable and equals a $Y' \in \bar{Y}$, then the conclusion holds because of $\vdash \bar{D}$.

2. By induction on T . The interesting case consists in (cf. the previous case) T or S being non-variable and containing parameters from $\bar{Y}$. Since T and S have the same structure same types from $\bar{Y}$ are substituted by corresponding types from $\bar{D}$ the conclusion holds because of SUB-CLASS.

3. By induction on M . The interesting cases are TERM-FIELD and TERM-METHOD. In the case of TERM-FIELD, the field typing $\bar{Y} \triangleleft \bar{C}; \bar{P} \bar{x} \vdash M.f_i : S_i$ can be derived from the hypotheses $\bar{Y} \triangleleft \bar{C}; \bar{P} \bar{x} \vdash M : T$ and $\bar{Y} \triangleleft \bar{C} \vdash \text{fields}(T) = \bar{S} \bar{f}$.

If T is ground, the conclusion can be reached using induction on T as in the previous cases and the property that subtyping preserves field lookup: if $\vdash T' <: T$ for well-formed T, T' and $\text{fields}(T) = \bar{V} \bar{f}$ then $\text{fields}(T') = \bar{V} \bar{f}, \bar{U} \bar{g}$ with $\bar{f}, \bar{g}$ disjoint.

If T is variable, $T = Y' \in \bar{Y}$ and the conclusion holds because of the preconditions by an induction as before and the property that subtyping preserves field lookup, see above.

4. By induction on the judgement typing L . The interesting case is TERM-PROCEED. In this case, $L = \text{proceed}(\bar{x})$ holds and $M : T'[\bar{D}/\bar{Y}]$ follows from $L[\bar{D}/\bar{Y}, \dots, \bar{M}/\text{proceed}(\bar{x})]$. Since also $M : S' <: S$ from one of the hypotheses of proceed substitutivity, $T' <: S$. Furthermore, $S' <: T \wedge T' <: T$ because M substitutes L . The conclusion then follows from the second substitutivity sublemma above. $\square$

Lemma 3.2. *For type environment Δ and term environments $\Gamma = \bar{P} \bar{x}$ and $\Gamma' = \bar{P}' \bar{x}$ such that $\Delta \vdash \bar{P}' <: \bar{P}$, it holds that, if $\Delta; \Gamma \vdash M : T$ then $\Delta; \Gamma' \vdash M : T'$ such that $\Delta \vdash T' <: T$.*

Proof. The proof is by induction on typing of M . The most interesting cases are TERM-FIELD and TERM-METHOD. In case of TERM-FIELD, the field typing $\Delta; \Gamma \vdash M.f_i : S_i$ can be derived from the hypotheses $\Delta; \Gamma \vdash M : T$ and $\Delta \vdash \text{fields}(T) = \bar{S} \bar{f}$. By using induction on T and the property that subtyping preserves field lookup, we have $\Delta; \Gamma' \vdash M : T'$ and $\Delta \vdash \text{fields}(T') = \bar{S} \bar{f}, \bar{V} \bar{g}$ with $\bar{f}$ and $\bar{g}$ disjoint. Applying TERM-FIELD provides $\Delta; \Gamma' \vdash M.f_i : S_i$. Case TERM-METHOD is similar, and uses the property that subtyping preserves method lookup. $\square$

Theorem 3.3 (Preservation). *If $\vdash M : T$ and $M \rightarrow M'$ then $\vdash M' : T'$ for some T' such that $\vdash T' <: T$.*

Proof. The proof is by induction on the evaluation rules. The most interesting case is EVAL-APPLY. From the induction hypothesis, the conditions of this evaluation rule, and the assumption that all declarations are well-typed, we have:

$$\vdash M.\ell < \bar{V} > [a, \bar{b}] (\bar{N}) : T \quad (3.1)$$

$$\vdash \text{advice } a < \bar{X} \triangleleft \bar{C} > R(\bar{P} \bar{x}) : \phi : S(\bar{Q}) \{L\} \quad (3.2)$$

$$\vdash \text{match}(M.\ell < \bar{V} > (\bar{N}), \phi) = \bar{y}; \bar{K}; I.h < \bar{U} > (\bar{J}) \quad (3.3)$$

From (3.1) and since TERM-ADVISED is the only rule to type this form, we have:

$$\vdash M.\ell < \bar{V} > (\bar{N}) : T \quad (3.4)$$

$$\mathcal{D} \ni \text{advice } b, \bar{a} \dots \quad (3.5)$$

From (3.2) and DEC-ADVICE, we have, with $\bar{Z}, W$ free variables:

$$\bar{X} \triangleleft \bar{C} \vdash \bar{C}, R, \bar{P}, S, \bar{Q} \quad (3.6)$$

$$\bar{X} \triangleleft \bar{C}; \bar{P} \bar{x}, S \text{ proceed}(\bar{Q}) \vdash L : R' \quad (3.7)$$

$$\bar{X} \triangleleft \bar{C} \vdash R' <: R \quad (3.8)$$

$$\vdash \phi : D-E(\bar{F}-\bar{G} \bar{z}) \quad (3.9)$$

$$\bar{X} \triangleleft \bar{C}, \bar{Z} \triangleleft \bar{G}, W \triangleleft E \vdash \bar{V}, \bar{V} <: \bar{C} \quad (3.10)$$

$$\bar{Z} \triangleleft \bar{G} \vdash \bar{x}[\bar{Z}/\bar{z}] <: \bar{P}[\bar{V}/\bar{X}] \quad (3.11)$$

$$\bar{Z} \triangleright \bar{F} \vdash \bar{Q}[\bar{V}/\bar{X}] <: \bar{x}[\bar{Z}/\bar{z}] \quad (3.12)$$

$$W \triangleleft E \vdash W <: S[\bar{V}/\bar{X}] \quad (3.13)$$

$$W \triangleright D \vdash R[\bar{V}/\bar{X}] <: W \quad (3.14)$$

From (3.3), (3.4) and (3.9) and assumption CONSISTENCY-PC, we have:

$$\bar{y} = \bar{z} \quad (3.15)$$

$$\vdash \bar{K} : \bar{G}' <: \bar{G} \quad (3.16)$$

$$\bullet; \bar{F} \bar{y} \vdash I.h < \bar{U} > (\bar{J}) : E' <: E \quad (3.17)$$

$$\vdash D <: T \quad (3.18)$$

We bind $\bar{Z}$ to $\bar{G}'$ in (3.11). Together with (3.16) and the substitution lemma, we obtain:

$$\vdash \bar{x}[\bar{K}/\bar{z}] : \bar{x}[\bar{G}'/\bar{z}] <: \bar{P}[\bar{V}/\bar{x}] \quad (3.19)$$

Next, we bind $\bar{Z}, W$ to $\bar{F}, E$ in (3.12) and (3.13). From the substitution lemma, we obtain:

$$\vdash \bar{Q}[\bar{V}/\bar{x}] <: \bar{x}[\bar{F}/\bar{z}] \quad (3.20)$$

$$\vdash E <: S[\bar{V}/\bar{x}] \quad (3.21)$$

We will construct the environment Γ' , that binds z_j to $Q_i[\bar{V}/\bar{x}]$ when $z_j = x_i$ or to F_j when z_j does not appear in $\bar{x}$. Because of (3.20), this is a stronger environment than $\bar{F} \bar{z}$ and we can apply lemma 3.2 to (3.17), and by using (3.21), (3.5) and TERM-ADVISED:

$$\bullet; \Gamma' \vdash I.\bar{h}\langle\bar{U}\rangle[\bar{b}](\bar{J}) : E'' <: S[\bar{V}/\bar{x}] \quad (3.22)$$

On (3.7), we can now apply the (proceed) substitutivity lemma with (3.19), (3.22) and (3.8), to obtain:

$$\vdash L[\bar{x}[\bar{K}/\bar{z}]/\bar{x}, I.\bar{h}\langle\bar{U}\rangle[\bar{b}](\bar{J})/\text{proceed}(\bar{x})][\bar{K}/\bar{z}] : R'' <: R'[\bar{V}/\bar{x}] <: R[\bar{V}/\bar{x}] \quad (3.23)$$

This is a typing for the evaluation result of rule EVAL-APPLY. Because of (3.18), we can bind W to T in (3.14) and obtain the required relation for the result of the typing:

$$\vdash R[\bar{V}/\bar{x}] <: T \quad \square$$

3.4.2 Progress

Theorem 3.4 (Progress). *If $\vdash M : T$ then either M is a value or $M \rightarrow M'$ holds for some M' .*

Proof. We consider each of the term typing rules that can establish $\vdash M : T$. TERM-VAR and TERM-PROCEED require a non-empty term environment, so they cannot establish this condition. In case of TERM-OBJECT, we immediately have that M is a value, or that there is progress for one of the arguments (which are well-typed). For TERM-FIELD, $M = I.f_i$ and the conditions will guarantee (through induction) that either I can be further evaluated or the rule EVAL-FIELD can be applied (this case is identical in FGJ). For TERM-METHOD, we can always apply the rule EVAL-SELECT.

The final case is TERM-ADVISED, where $M = I.\ell\langle\bar{V}\rangle[\bar{a}](\bar{N})$. In case $\bar{a}$ is non-empty, the conditions of the rule will guarantee that there exists a top-level declaration of a_1 (which we assume to be well-typed). In section 3.2.1 we require for its pointcut that either the subterms I or $\bar{N}$ can be further evaluated (i.e. there is progress in a congruent evaluation context), or the pointcut matches (in which case we can apply EVAL-APPLY) or it does not match (in which case we can apply EVAL-NOAPPLY). In case $\bar{a}$ is empty, we have reached the actual execution of the method call. The first condition of the rule TERM-ADVISED establishes that $I.\ell\langle\bar{V}\rangle(\bar{N})$ is well-typed, and the conditions of TERM-METHOD will therefore guarantee (through induction) that either I can be further evaluated or EVAL-METHOD can be applied, identical to how progress is guaranteed for a well-typed method call in FGJ. $\square$

Chapter 4

AspectJ-like pointcuts

To illustrate how the AspectJ pointcut language can be integrated in the proposed framework, we will now concretize the abstract pointcuts from the previous sections. We employ a pointcut form that is a fixed conjunction of three parts; we believe this form is illustrative for the capabilities of the AspectJ pointcut language, while keeping the presentation relatively simple (an integration of the full pointcut language would, for example, require the usage of a boolean algebra of bindings, to accomodate for the free combination of pointcuts using logical operators). The pointcut form is the following:

$$\phi ::= \theta \ \&\& \ \mathbf{target}(F-G \ y) \ \&\& \ \mathbf{returning}(D-E)$$

Here, θ represents a selection pointcut that is still kept abstract. It models the many AspectJ pointcut forms that merely pick out join points (i.e. that do not organize passing of parameters between join point and advice), such as the `execution` and `within` primitives. These forms often match the compile-time signature of the join points against given method or type patterns. Although the terms and evaluations rules would need to be modified to carry the initial types throughout the evaluation in order to support this, this is straightforward and it does not contribute to the main objective of this framework, namely the verification of parameter passing between join point and advice. We therefore believe it is justified to keep these selection pointcuts abstract.

The `target` primitive, in turn, is very similar to its AspectJ counterpart: it exposes the receiving object of a method invocation and it only matches join points where this receiving object is of an appropriate type. Finally, the `returning` primitive does not exist in AspectJ, but it is conceived similarly to `target`: it only matches join points where the return type is appropriate.

4.1 Matching and Typing Judgements

The matching semantics of this pointcut form is formally defined by the following rule:

$$\begin{array}{c} \text{MATCH-PC} \\ \frac{\begin{array}{l} \vdash M.\ell\langle\bar{V}\rangle(\bar{N}) \text{ selected by } \theta \quad M = \mathbf{new} \ C(\dots) \quad \vdash F <: C <: G \\ \vdash \mathbf{meth}(C.\ell) = \dots R(\dots) \dots \quad \vdash D <: R <: E \end{array}}{\vdash \mathbf{match}(M.\ell\langle\bar{V}\rangle(\bar{N}), \theta \ \&\& \ \mathbf{target}(F-G \ y) \ \&\& \ \mathbf{returning}(D-E)) = y; M; y.\ell\langle\bar{V}\rangle(\bar{N})} \end{array}$$

Three main conditions have to be satisfied for the pointcut to match. First, the selection pointcut θ must match the join point, as verified through the judgement `selected by` (which is kept

abstract). Second, the dynamic type of the receiver must lie within the type range given as an argument to the **target** primitive. Third, the return type of the method lookup must lie within the type range given as an argument to the **returning** primitive.

Since the **target** and **returning** primitives only select join points with appropriate types, the typing of the pointcut is derived in a straightforward manner from the types given as arguments to these primitives:

TYPE-PC

$$\frac{}{\vdash \theta \ \&\& \ \text{target}(F-G \ y) \ \&\& \ \text{returning}(D-E) : D-E(F-G \ y)}$$

Caveat Informally, we can say that a pointcut with this type can be bound to an advice with signature $D(G \ y)$ (or stronger) and proceed signature $E(F)$ (or weaker). While the relations $\vdash F <: C <: G$ and $\vdash D <: R <: E$ will guarantee correct argument passing between join point and advice (as shown in the next section), the matching is overly selective in some cases. As an example, consider the term p , where p is a variable that has type **Person** in the accompanying type environment. A well-typed method call on p can only invoke the methods defined for **Person** (or its superclasses). So it is in fact safe to have the pointcut **target(Person-Person)** match these calls: the exposed receiving object will always be a **Person**, and it can be replaced with any **Person**, since any **Person** will understand the method call. However, if the variable p is bound to a value **new Employee()** during the evaluation, then its dynamic type no longer meets the lower bound of the type range, and the join point will not be selected. Again, knowledge of the initial type of a term would thus be required in order to avoid this restriction.

4.2 Consistency Property

We can then show that matching and typing, as we defined them, are consistent in the manner required by the assumption CONSISTENCY-PC. This assumption is the only requirement regarding pointcuts in the proof of the preservation theorem.

Proof. The name of the exposed argument is y in both MATCH-PC and TYPE-PC, this establishes the first conclusion of CONSISTENCY-PC. The exposed argument itself is M , which the second condition of MATCH-PC requires to be of the form **new** $C(\dots)$. Given that $M.\ell < \bar{V} > (\bar{N})$ is well-typed, we know from TERM-METHOD that M is well-typed, so from TERM-OBJECT, we have:

$$\vdash M = \text{new } C(\dots) : C$$

This establishes the second conclusion of CONSISTENCY-PC, while the third condition of MATCH-PC establishes the third conclusion:

$$\vdash C <: G$$

The proceed term is $y.\ell < \bar{V} > (\bar{N})$. From the fourth condition of MATCH-PC and the relation $\vdash F <: C$ from the third condition, we have¹:

$$\begin{aligned} \vdash \text{meth}(F.\ell) &= \dots R'(\dots) \\ \vdash R' &<: R \end{aligned}$$

TERM-VAR immediately establishes $\bullet; F \ y \vdash y : F$. Given that $M.\ell < \bar{V} > (\bar{N})$ is well-typed, and using TERM-METHOD, we can use the above to type the proceed term:

$$\bullet; F \ y \vdash y.\ell < \bar{V} > (\bar{N}) : R'$$

¹Using the lemma “Subtyping preserves method lookup”

This establishes the fourth conclusion of CONSISTENCY-PC. The fifth conclusion can be established using the fifth condition from MATCH-PC:

$$\vdash R' <: R <: E$$

Given that $M.\ell < \bar{V} > (\bar{N})$ is well-typed, we know from TERM-METHOD and the fourth condition of MATCH-PC that its type will be R . The fifth condition of MATCH-PC therefore directly establishes the sixth and last conclusion of CONSISTENCY-PC:

$$\vdash D <: R \quad \square$$

Chapter 5

Conclusion

In this technical report, we have briefly reviewed the type safety problems of pointcut/advice bindings in AspectJ-like languages and previous (partial) solutions to these problems that have been proposed in the literature. We have then presented a formal semantics and a type system for an extension of AspectJ that addresses open issues with the correct typing of aspects in such languages.

Our type system extends these previous approaches by, in particular, ensuring safety for both generic and non-generic pointcut/advice declarations. Pointcuts quantify over heterogeneous sets of join points and are hence typed using type ranges in our approach, while type variables and a separate signature for **proceed** declarations allow to express the generic and invasive nature of advice. Using these mechanisms, we can express advice that augments, narrows or replaces base functionality in possibly generic contexts. We have presented a complete formal account of our approach by appropriate evaluation rules, typing rules, type preservation and progress theorems as well as corresponding proofs.

Bibliography

- [1] Curtis Clifton and Gary T. Leavens. MiniMAO1: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63(3):321–374, 2006.
- [2] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proc. of OOPSLA '99*, pages 132–146. ACM Press, October 1999.
- [3] Radha Jagadeesan, Alan Jeffrey, and James Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 63(3):267–296, 2006.
- [4] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In *Proc. of Foundations of Object-Oriented Languages (FOOL 2005)*, January 2005.
- [5] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal M. Gafter. Adding wildcards to the Java programming language. In Hisham Haddad, Andrea Omicini, Roger L. Wainwright, and Lorie M. Liebrock, editors, *Proc. of the 2004 ACM Symposium on Applied Computing (SAC)*, pages 1289–1296. ACM Press, 2004.
- [6] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(5):890–910, 2004.