StrongAspectJ: Flexible and Safe Pointcut/Advice Bindings *

Bruno De Fraine[1] † Mario Südholt[2] Viviane Jonckers[1]

[1]System and Software Engineering Lab (SSEL); Vrije Universiteit Brussel; Belgium
[2]OBASCO project; École des Mines de Nantes-INRIA, LINA; France
bdefrain@vub.ac.be, sudholt@emn.fr, vejoncke@ssel.vub.ac.be

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 severely restrict the definition of certain generic advice behavior.
In this paper, we analyze the roots of these type errors, and describe measures to recover type 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 dual advice signature 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. As a language engineering contribution, we integrate our proposal with the AspectJ language, and we provide a prototype implementation as a plugin for the AspectBench Compiler (abc). On a theoretical level, we present a formal definition of the proposed constructs and typing rules, and develop proofs for their type safety properties.

1. Introduction

Aspect-oriented software development aims to solve crosscutting concerns by extending, often object-oriented, base languages with specific language constructs to enable the clean modularization of these concerns. One of the common design goals of such language extensions is a seamless integration with the base language. To this end, the aspect language will adopt a maximum of characteristics from the base language. With respect to typing, aspect-oriented extensions of statically-typed languages, such as AspectJ [19] for Java, aim to introduce statically-typed aspect modules.

However, AspectJ does not succeed in carrying over Java's static type-safety. Whereas a runtime type error can only originate from an explicit (narrowing) cast introduced by the programmer in Java, there is no such guarantee in AspectJ: as was previously identified [31, 6], the type variance rules for pointcut and advice declarations make the invocation of the *proceed*-method inherently unsafe. Furthermore, despite AspectJ 5's adoption of *Generics* [9], support for typing generic advice methods is limited. For these cases, AspectJ provides only an ad hoc solution that introduces new safety problems in the advice application [17, 23]. We note that this problem is not limited to AspectJ. To our knowledge, all other typed aspect languages for Java (such as Caesar [26] or JAsCo [29]) also suffer from these problems as they, in best case, adopt the same typing strategy as AspectJ.

In this paper we propose a novel type system that recovers safety for pointcut and advice declarations. Its main design goal is to support most common aspect definitions, including both generic and non-generic advice behavior. Furthermore, the elements of the type system can be integrated with practical aspect languages, and we present such an integration for AspectJ.

Contributions. The concrete contributions of this paper are the following:

- We present an analysis of the current typing rules for pointcut and advice declarations, and show how they can cause different classes of type errors that remain undetected at compile-time.
- We propose typing mechanisms that avoid these problems; they include a dual advice signature for *around*-advice, type ranges to describe the (possibly heterogeneous) selection made by pointcuts, and type variables for generic advice behavior. We derive type relations for these elements to guarantee safe advice application.
- We present the StrongAspectJ language, an extension of AspectJ that incorporates our type system. An implementation of StrongAspectJ is provided as a plugin for the AspectBench Compiler (*abc*) [2]. Furthermore, we show that our typing rules can also be enforced in framework-based AOP approaches that employ only a generics-aware base language compiler.
- We provide a formal definition of the syntax, matching semantics and typing rules of the proposed extensions. Proofs of type safety properties are sketched in an appendix, and more details are available in a companion technical report [10].

While previous theoretical research [31, 17, 6, 23] has sporadically identified the deficiencies discussed in this work, we present a categorization of the different problems through a comprehensive discussion of the current type system. Compared to their proposals, we provide clear extensions and we present concrete integration in mainstream approaches (such as AspectJ and AOP frameworks), an issue that has not been previously addressed.

Paper Structure. Section 2 analyzes the typing rules of current aspect languages, and presents the typing problems that motivate this paper. Section 3 informally presents our type system: the underpinning typing principles are explained in a general manner, and

* This work has been supported by AOSD-Europe, the European Network of Excellence in AOSD (`aosd-europe.net`).

† The author is supported by a doctoral grant from the Institute for the Promotion of Innovation through Science and Technology in Flanders.

AOSD'08, March 31 – April 4, 2008, Brussels, Belgium.
Copyright © 2008 ACM 978-1-60558-044-9/08/0003…$5.00

the concrete StrongAspectJ and AOP framework realizations are presented. Some representative examples of realistic advice behavior are developed using the new constructs in section 3.2.3. In section 4, we discuss the *abc* implementation of the StrongAspectJ language, and in section 5, we give a formal definition of the key elements of our proposal. Finally, we present some related work and state our conclusions in sections 6 and 7.

2. Motivation

In this section, we recapitulate the current AspectJ typing rules and outline the type-safety problems we have identified. These issues relate to the central pointcut/advice mechanism which allows aspects to intercept join points in the execution of an application.

In what follows, we will use the term "signature" in a generalized meaning to denote a contract stipulating parameter types and a return type. While the concept of a signature is widely used for ordinary methods and advice methods, and while the declaration of a named pointcut also specifies a signature, we extend this usage to join points as they are captured by a pointcut. For example, the execution join point of method `Person.getAge()`, has signature `Integer(Person p)` when it is selected by the following pointcut (i.e. it can be executed with an argument `p` of type `Person`, and it will return a value of type `Integer`):

```
execution(Integer Person.getAge()) && this(p)
```

Additionally, we consider a partial order relation between signatures to indicate when the contract of one signature provides stronger guarantees than another signature: we say signature A is *stronger* than signature B, when the return type of A is a *sub*type of the return type of B, and when the argument types of A are *super*types of the respective argument types of B. The different treatment of argument and return types is referred to as the principle of *contravariance* for arguments, and *covariance* for results. For example, the signature `Integer(Person p)` is stronger than the signature `Number(Employee p)`, because it requires a less specific argument and promises a more specific return value. (Here, and in the rest of the examples throughout the text, we use the well-known type hierarchy `Employee` $<:$ `Person` $<:$ `Object` for argument types, and `Integer` $<:$ `Number` $<:$ `Object` for return types, where "$<:$" indicates "is a subtype of".) Generally, a method or join point can be substituted in place of another one, if the signature of the first is equal to or stronger than the signature of the latter. This ordering is therefore also known as the subtype relation for function types in some literature.

2.1 Current typing rules and subtype variance

In current typed aspect languages, the typing of pointcut and advice declarations is typically founded on the following principles: (i) the body of an advice method must adhere to the advice signature (identical to how a regular method must adhere to its signature), (ii) the pointcut signature must be stronger than the signature of the join points that it selects, i.e. the selected join points must adhere to the pointcut signature and (iii) when an advice is bound to a pointcut, the signature of the advice must be stronger than that of the pointcut. Together these principles ensure that the advice signature will be stronger than the signature of an intercepted join point, and the advice code can therefore be safely executed in addition to, or in place of, this join point.

As a concrete example, consider the following valid declarations in AspectJ.

```
pointcut pc(Employee e): args(e,..) && within(Main);
before(Person p): pc(p) { /* ... */ }
Integer around(): call(Number *()) { /* ... */ }
```

The named pointcut in the first line declares an argument of type `Employee` in its signature, and hence the pointcut expression must always bind an argument of this type. This can be accomplished with the primitives **`this`**, **`target`** and **`args`**, which bind respectively the executing object, the receiving object or the arguments of the join point to a given variable, but only match if the to-be-matched object belongs to the variable's type (or its subtypes). Because of this dynamic type test, the pointcut signature will be equal or stronger than the join point signature, as its argument will be of an equal or wider type (recall from the above that a stronger signature implies wider argument types). The signature of the advice in the declaration on the second line is even stronger, since the argument type is further widened to type `Person`. The binding of this advice to the pointcut `pc` is therefore accepted.

Return types on the other hand, are only declared for *around*-advice kind in AspectJ. Since this advice is executed *in place of* the join point, it needs to provide a return value to return to the join point caller. Similar to the argument types, the return type of an *around*-advice has to be verified against the pointcut. However, AspectJ does not include a return type in pointcut signatures. The verification of the return type is instead postponed until weaving the advice, when the join point shadows are determined, and their return types can be taken into account[1]. Nonetheless, the compatibility of the advice is still determined according to the principle that the advice signature should be equal or stronger than the join point signature, i.e. that the advice return type should be equal or narrower. Since the advice declaration in the third line only applies to join points with exactly the static return type `Number`, the advice return type (`Integer`) will always be narrower, and the advice will always be accepted.

2.2 Around advice and the proceed mechanism

In the body of an *around*-advice, a *proceed*-method can be employed to invoke the execution of the intercepted join point (or to call the next advice in an advice chain), possibly with different parameter values. Since the *around*-advice acts effectively as a wrapper around the join point in this manner, we can identify two interfaces associated with this advice: the *proceed* interface is fulfilled by the intercepted join point and offered to the advice by means of the *proceed*-method, while the interface of the advice method itself is offered by the advice to the caller of the join point. Put differently, these are respectively the *expected* and *provided* interfaces of an *around*-advice.

However, current typed aspect languages do not distinguish between these two interfaces when determining the signature of the *proceed*-method. For example, in AspectJ, the *proceed* signature is taken to be the same as the advice signature [19, sec. 3.6], which amounts to ruling that — in the sense of these interfaces — the advice can expect the same as what it provides. This is an incorrect judgment in the cases of type variance explained in the previous section (and already discussed in a more limited context in [31, 6]), as the advice signature can be stronger than the join point signature, and the advice then provides more than it can expect.

As a concrete example, the following advice is accepted by AspectJ's typing rules:

```
void around(Person p):
  execution(void *()) && this(p) {
    proceed(new Person());
}
```

[1] In case of compile-time weaving, this might still qualify as a static verification. Note however that the pointcut/advice compatibility has become dependent on the particular base application on which the advice is applied, an undesirable property.

While this advice works correctly with any Person as an argument, it may incorrectly assume the same of all of its join points. The advice can also be applied to methods of subclasses, e.g. Employee.promote(), in which case executing the join point via the *proceed*-method with an argument of the general type Person will cause a ClassCastException.

An identical situation can also occur with respect to the return types, as demonstrated by the following advice:

```
Integer around(): call(Number *()) {
    Integer i = proceed();
    // ...
}
```

Because this advice provides an Integer result, it can assume that its join points do so as well in AspectJ. However, the advice can also be applied to join points where the returned value of type Number is not an Integer, in which case a ClassCastException will occur when returning from the *proceed*-method to the advice body.

The type systems of current aspect languages (such as AspectJ) do not prevent these type errors at compile- or weave-time. They are caught by the runtime type checks of the execution platform when the advices are executed with an incompatible context. Such errors can be difficult to detect as the problem might only manifest itself in an uncommon situation or after the base program has evolved in a certain way (when, for example, new subclasses have been introduced).

One obvious measure to prevent these type errors would be to prohibit the corresponding forms of type variance. Alternatively, the *proceed*-method could be defined as not taking arguments and thus always invoke the join point with the original arguments (plus a similar restriction for the return value). However, both modifications would clearly be very restrictive.

2.3 Special case of the Object return type

Of course, not every advice will use the full replacement power enabled by *around*-advice. Some advices will always invoke the *proceed*-method with an original argument value from the join point caller, or will always return a value obtained from the join point through a *proceed* invocation. Since these values will always be of a correct type, the advice is generally compatible with any join point (save other assumptions about these values). In this sense, the advice is *generic* with respect to its return value or arguments. For example, the following advice, which executes the intercepted join point twice and which returns the result of the last invocation, is generic with respect to its return value and is therefore compatible with any join point return type.

```
Object around(): call(* incr()) {
    proceed(); return proceed();
}
```

However, note that this advice binding is not as such allowed by the type variance principle from section 2.1, which states that the advice signature has to be equal or stronger than the join point signature, i.e. that the advice return type has to be narrower than the join point return type (while Object will generally be wider instead). Since it is quite common and useful for an advice not to interfere with the execution of the base functionality (e.g. in common AOP applications, such as a profiling aspect), this is a severe restriction. AspectJ therefore employs an additional type variance rule to accommodate for generic advice of this kind: when the return type of an *around*-advice is declared as java.lang.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.

Alas, this condition is neither necessary nor sufficient, and this causes further type errors on the one hand, and prevents the typing of certain valid advice methods on the other hand. For example, the following advice declares the Object return type, but is by no means generic:

```
Object around(): call(Number *()) {
    return new Object();
}
```

Since the advice does not have to follow the type variance principle from section 2.1, it can return a value which cannot be handled by the corresponding base functionality, which will thus cause a ClassCastException (as also observed, though without relation to the type variance principle, in [17, 23]).

As an example of a case of generic advice that uses a type different from Object, consider the following generic advice method that is in fact sound for any join point with a return type that is a subtype of Number. However, as the advice is not declared with the Object return type, the opposite relation (supertypes of Number) is enforced by AspectJ. The following pointcut binding is thus illegal:

```
Number around(): call((Integer || Float) *(..)) {
    Number n = proceed();
    while(n.intValue() > 100)
        n = proceed();
    return n;
}
```

The workaround to enable this binding consists in declaring the Object return type for this advice and including explicit casts to Number at every invocation of the *proceed*-method. This is tedious, and of course it does not guarantee static type safety either.

3. Safe and Flexible Pointcut/Advice Bindings

In this section, we informally present our proposed solution for the typing problems from the previous sections. We first explain the general principles for typing pointcut and advice declarations, and then develop StrongAspectJ, a full integration with the concrete AspectJ language. After presenting some typical examples of advice behavior expressed using StrongAspectJ, we outline an alternative realization of the typing principles as a type system for framework-based AOP approaches such as Spring AOP or JBoss AOP.

3.1 Typing principles

To present our proposal at an abstract level, we will develop replacements for the typing principles outlined in section 2.1.

3.1.1 Dual advice signature

As described in section 2.2, the behavior of an *around*-advice is 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 propose to use an explicit *proceed*-signature in addition to the regular advice signature for *around*-advice in order to reflect 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.

3.1.2 Type variance relations, pointcut type ranges

This dual signature has to be taken into account when verifying the compatibility of the advice with a pointcut and, ultimately, its join points. To derive the necessary relations, we observe that both the join point and its caller are unmodified by the advice

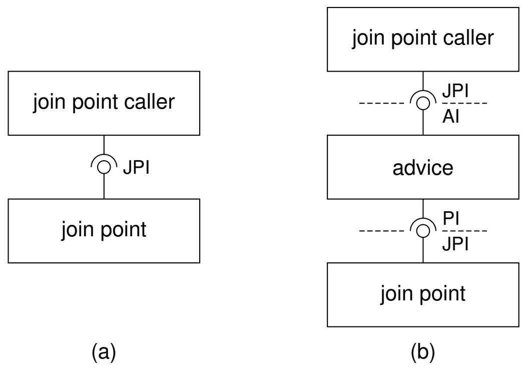

Figure 1. Join point interfaces (a) without and (b) with advice application, in UML ball-and-socket notation [25]

application, and thus rely on the original join point interface for their interaction with the advice (see figure 1). Consequently, the advice interface (AI) is provided where the join point interface (JPI) is expected, and the join point interface is provided where the proceed interface (PI) is expected. In order to obtain a safe advice application, we accordingly require an advice signature which is equal or stronger than the join point signature (i.e. the original relation from section 2.2), as well as a join point signature which is equal or stronger than the *proceed* signature, that is, we impose a new relation that harnesses the new interface.

These variance relations 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). For example, when a pointcut is typed with the signature range `Integer-Number(Employee-Person p)`, then its join points must have a signature that is no weaker than `Number(Employee p)` (the upper bound with respect to signature strength), and no stronger than `Integer(Person p)` (the lower bound).

We can then enforce the relations of the previous paragraph when verifying a pointcut/advice binding, by requiring that the advice signature is stronger than the lower bound of the pointcut signature, and that the upper bound of the pointcut signature is stronger than the *proceed* signature. Or, to summarize the situation in a mathematical notation (where "σ_x" represents "signature of x" and "$\leq$" represents "is equal or stronger"):

$$\sigma_{advice} \leq \sigma_{pc,lower} \leq \sigma_{jp} \leq \sigma_{pc,upper} \leq \sigma_{proceed}$$

We can group these relations by considering *containment* of ranges: a range *contains* another range if it has both an equal or more specific lower bound and an equal or more general upper bound. So (where "$\in$" represents "lies within" and "$\subseteq$" represents "is contained by"):

$$\sigma_{jp} \in [\sigma_{pc,lower} - \sigma_{pc,upper}] \subseteq [\sigma_{advice} - \sigma_{proceed}]$$

3.1.3 Advice type variables

Finally, similar to other proposals [17], we propose to use type variables at the level of advice to support the generic advice behavior of section 2.3 in the type system[2]. A type variable represents an unknown type, possibly below a given non-variable bound, which can be freely instantiated for each advice application. A typing using an unknown type X is a stronger guarantee than using `Object` (or using its upper bound): while any value of type X can be assigned to a variable of type `Object` (or of its upper bound type), the opposite does not hold. But as generic advice behavior always invokes the *proceed*-method with an original argument (or it always returns a result obtained from a *proceed* invocation), it is generally able to keep the type of this argument (or of the result) unknown. The bound of the type variable can be used to represent additional assumptions made by the advice about this unknown type.

Which pointcuts can then be safely bound to an advice typed with type variables? While the conclusions of the previous section stay in effect, the ability to freely bind type variables for each advice application makes it easier to meet the required relations. We observe that, when a type variable appears as the type of the return value (resp. argument) in both advice and *proceed* signatures, and it does not appear elsewhere in these signatures, we can bind this type variable to the type of the return value (resp. corresponding argument) in the join point signature (at least, if this binding is allowed by the type variable's bound). As such, we will have obtained that the advice signature, *proceed* signature and join point signature all have the same type (i.e. no variance) for this return value or argument, and thus satisfy the previous relations. We will therefore allow the binding of this advice to pointcuts whose upper bound for the return value (resp. corresponding argument) is below the upper bound of the type variable.

3.2 StrongAspectJ

We now present StrongAspectJ[3], an integration of the proposed typing principles with the AspectJ language. We first discuss its syntax, matching semantics and typing rules in a general manner, and then present two sets of examples of advice behavior expressed in StrongAspectJ.

3.2.1 Language definition

Our proposed extensions to AspectJ are specified in figure 2, which contains a formal definition of the relevant parts of the language syntax, and in figure 3, which lists the typing and matching rules for the redefined constructs in a systematic (but still informal) manner.

Notational conventions. The definition borrows some notational conventions from the Featherweight Java calculus [16]. 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 under the heading "Names and variables". Different metavariables are used for the same syntactic category, in order to distinguish between multiple instances of the same syntactic element in one definition phrase. Additionally, we write $\bar{e}$ for an ordered sequence of zero or more elements $e_1, \ldots, e_n$, 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. $\bar{C}\ \bar{x}$ signifies $C_1\ x_1, \ldots, C_n\ x_n$. An individual element is referred to as e_i. A form of syntactic sugar not shown in the syntax definition, is the possible omission of the empty angle brackets "<>" when no type variables are declared or when no type arguments are provided.

[2] The usage of type variables for the typing of generic (or *parametrically polymorphic*) behavior is wide-spread [5], and exists at the level of methods and classes in Java 5 [24].

[3] The "strong" prefix refers to the advanced safety guarantees provided by the extension. The name is also loosely inspired by the Strongtalk type system for Smalltalk [3].

// Names and variables

c, d		*// Class name*
p		*// Pointcut name*
x, y, z		*// Term variable*
X, Y, Z, W		*// Type variable*

// Type-related categories

C, D, E, F, G	$::=$	$c\texttt{<}\bar{T}\texttt{>}$ *// Non-variable type*
P, Q, R, S, T	$::=$	$X \mid C$ *// Type (var or non-var)*

// Terms and pointcut expressions

M, N, I, J, K	$::=$	$\ldots \mid$ **proceed**$(\bar{N}) \mid \ldots$ *// Term*
ϕ	$::=$	$\ldots$ *// Pointcut expression*
	$\mid$	$p(\bar{x}) \mid$ **this**$(E\text{-}F\ x)$
	$\mid$	**target**$(E\text{-}F\ x) \mid$ **args**$(\bar{E}\text{-}\bar{F}\ \bar{x})$

// Top-level declarations

$\mathcal{D}$	$::=$	**pointcut** $p(\bar{E}\text{-}\bar{F}\ \bar{x})$: ϕ;
	$\mid$	**before**$(\bar{F}\ \bar{x})$: ϕ $\{M\}$
	$\mid$	**after**$(\bar{F}\ \bar{x})$: ϕ $\{M\}$
	$\mid$	$\texttt{<}\bar{X}$ **extends** $\bar{C}\texttt{>}$ R **around**$(\bar{P}\ \bar{x})$: ϕ: S **proceed**$(\bar{Q})$ $\{M\}$

Figure 2. StrongAspectJ syntax (relevant parts)

Around advice declaration (cf. the last phrase of $\mathcal{D}$ in fig. 2)

Type use: Type variables $\bar{X}$ may be used as subtypes of bounds $\bar{C}$ in the entire advice declaration. All types must be known type variables or legal parametrized types as per [13, §4.5].

Variable signature types: Each type variable X_i may be used at most once in the signature types $\{R, \bar{P}\}$ and in $\{S, \bar{Q}\}$. X_i must then appear in the same position in both signatures.

Parameter use: Identifiers $\bar{x}$ may be used as simple names in body M to refer to parameter variables of declared types $\bar{P}$.

Proceed use: Fixed identifier **proceed** may be used as a simple method name with declared signature $S(\bar{Q})$ in body M.

Body return value: When declared with non-**void** return type R, it is not allowed to drop off the end of the body M, and every **return** must have an expression of some subtype of R.

Pointcut usage (cf. the first and last phrase of $\mathcal{D}$ in fig. 2)

Parameter binding: Declared advice or named pointcut parameters $\bar{x}$ must be bound (i.e. used as an argument) in each disjunctive branch of employed pointcut expression ϕ.

Pointcut parameter type: When named pointcut parameter x_i is used as argument in a pointcut expression, the argument position type range must be contained in type range $E_i\text{-}F_i$.

Advice parameter type: When *around* advice parameter x_i is used as argument in a pointcut expression, the argument position type range must be contained in range $Q_i\text{-}P_i$, if P_i and Q_i are non-variable, or must lie below C_k, if P_i and Q_i both equal type variable X_k with bound C_k.

Advice return type: The compile-time return type of join points where *around* advice is applied, must lie within advice return types $R\text{-}S$, if R and S are non-variable, or must lie below C_k, if R and S both equal type variable X_k with bound C_k.

Pointcut matching (cf. ϕ in fig. 2)

Binding primitives: Primitive pointcuts **this**, **target** and **args** match if the compile-time type of the to-be-bound variable or expression is some supertype of the declared lower bound E, and the run-time type is some subtype of upper bound F.

Figure 3. StrongAspectJ semantics: typing and matching rules

Overview. The syntax specification in figure 2 consecutively defines the structure of types (general, variable and non-variable), intermediate expressions (terms and pointcuts), and top-level declarations (named pointcuts and advice). The parts omitted from this figure keep their original AspectJ definition. In figure 3, the first section of typing rules regulates the use of types and terms in the signature(s) and body of *around* advice declarations. The second section treats the usage of pointcut expressions in the declaration of named pointcuts and advice (i.e. the pointcut/advice binding). The last section defines the matching rules for pointcut expressions; here only the primitives **this**, **target** and **args** are redefined.

Application of typing principles. The StrongAspectJ extension is the result of the integration of the principles of section 3.1 with the concrete elements of AspectJ and Java. As proposed in section 3.1.3, we allow the introduction of type variables as a part of the *around*-advice declaration with standard usage constraints (rule *Type use*). *Around*-advice also includes the additional *proceed* signature from section 3.1.1, and this declaration is enforced in the body (rule *Proceed use*). The rule *Variable signature types* requires that each type variable can be used at most once in these two signatures, in the same position, following the observation from section 3.1.3 that this measure allows the type variable to 'capture' the corresponding join point type (if this type does not exceed the type variable bound).

Moreover, the principles from section 3.1.2 are applied: type ranges are employed for the typing of pointcuts, and pointcut bindings must honor the stipulated type range relations (see the three last typing rules from section *Pointcut usage* in figure 3). The advice rules also include a case for variable signature types, where the variable's bound must be observed instead (section 3.1.3). The binding rules for *before* and *after* advice are not listed in figure 3, but stay in effect from AspectJ, with the clarification that the pointcut type must be abstracted to the lower bound signature. Additionally, three design decisions require more explanation:

1. We increase the expressive power by including `Null` (in addition to `Object`) as a built-in non-variable type. `Null` (the type of **null**) is at the bottom of every type hierarchy: it is a subtype of every type[4]. By employing `Null` as a lower bound in a type range, the range becomes unbounded in that direction[5].
2. The binding primitives **this**, **target** and **args** now match join points where the to-be-bound value has a type within the variable's declared type range, but notice from the matching rule that the upper bound is compared against the *dynamic* type of the join point value (a relaxation to select more join points). Also, unlike AspectJ, the variable's type range is specified inline in order to make the pointcut expression independent of the enclosing definition.
3. Similar to AspectJ, StrongAspectJ only declares arguments type ranges in the definition of named pointcuts (first phrase of $\mathcal{D}$) and verifies return types while weaving *around* advice (rule *Advice return type*). The return type is not relevant for the other advice kinds, and its inclusion would complicate the definition of named pointcuts that are never used for *around* advice. (This is a trade-off: in the framework-based implementation of our approach in section 3.3, as well as in the formal type system in section 5, we make pointcut return types explicit.)

[4] Although this type is explicitly considered by the Java language specification [13, §4.1], it is not allowed to use `Null` as a type annotation in Java.

[5] We observe that type ranges thus become a generalization of the wildcards for type arguments in Java 5 Generics [30]:

`<?` **extends** C`>`	reduces to	`Null`-C
`<?` **super** C`>`	reduces to	C-`Object`

3.2.2 Examples revisited

We now revisit some of the examples from section 2 to explain the StrongAspectJ typing rules, and to illustrate how they resolve the identified AspectJ typing problems. The relevant rule(s) are mentioned between parentheses at the beginning of each example.

Example (Pointcut matching). The expression

```
args(Null-Employee e,..)
```

selects join points where the first argument has a dynamic type `Employee`, or belongs to one of its subtypes. This is the semantics of the original **args** primitive with `Employee` as the declared parameter type. The pointcut `this(Employee-Person p)` matches join points belonging to the classes `Employee` and `Person` and all classes between them in the inheritance hierarchy.

Example (Pointcut parameter type). The following pointcut declarations are legal because the pointcut argument type ranges are only widened. In contrast, a pointcut parameter declared with type range `Employee-Employee` cannot be bound to the argument of these pointcuts.

```
pointcut pc0(Person-Person p): this(Person-Person p);
pointcut pc1(Employee-Person p): pc0(p);
pointcut pc2(Employee-Object x): pc1(x);
```

Example (Proceed use and Advice argument type). The following *around*-advice proceeds the intercepted join point with a new `Person` instance. It can be validly bound to a pointcut that matches join points in the class `Person` (strictly). To bind it to a pointcut that matches join points in subclasses, the argument of the *proceed*-signature must be narrowed (e.g. to `Employee`). However, then the invocation of the *proceed*-method with a general `Person` becomes illegal. As such, the type errors of section 2.2 can be prevented.

```
void around(Person p):
  execution(void *()) && this(Person-Person p):
  void proceed(Person) {
    proceed(new Person());
}
```

Example (Body return value and Advice return type). Rounding advice that returns the integer value of the original return value can be bound to a pointcut that selects join points with a static return type between `Integer` and `Number`. Should weaving occur at a join point with a return type of `Float` or `Object` (or a type unrelated to `Number`), a compile error is raised.

```
Integer around():
  call((Integer || Number) *()):
  Number proceed() {
    return new Integer(proceed().intValue());
}
```

Example (Advice parameter type and Variable signature types). The following advice executes the intercepted join point twice with the original parameter value. It can be bound to any pointcut with one argument of any type (represented here by the most general type range `Null-Object`).

```
abstract pointcut twice(Null-Object x);

<T> void around(T t): twice(t): void proceed(T) {
    proceed(t); proceed(t);
}
```

`<?>`	reduces to	`Null-Object`
C	reduces to	C-C

```
aspect NumberCache perthis(cachedOp()) {
    pointcut cachedOp():
      execution(Number DataProvider.expensive());
    Number cache;
    Number around(): cachedOp(): Number proceed() {
        if(cache == null)
            cache = proceed();
        return cache;
    }
}
```

Listing 1. Example Caching Aspect

Example (Advice return type and Type use). By declaring a variable bound, the available interface can be made more specific and the following advice can be admitted. In this case, the join points matched by the bound pointcuts must have a static return type that is a subtype of `Number`.

```
<N extends Number> N around():
  call((Integer || Float) *(..)):
  N proceed() {
    N n = proceed();
    while(n.intValue() > 100)
        n = proceed();
    return n;
}
```

3.2.3 Some (more) realistic examples

In this section, we present a number of examples of common aspect applications expressed using StrongAspectJ. Beside enabling a better understanding of the proposed typing constructs, these examples also illustrate the usefulness of the introduced mechanisms for practically-relevant, realistic advice behavior. They provide a reasonable indication that our typing schemes do not significantly restrict AspectJ's expressiveness. To categorize the demonstrated advice, we use the terminology of [28]. This work distinguishes between *augmentation* advice (which always executes the original behavior entirely), *narrowing* advice (which either executes the original behavior or raises an error) and *replacement* advice (which replaces the original behavior with entirely new behavior).

Caching is a common example of a concern that can be implemented using aspects (e.g. [7]). In listing 1, we show a simple caching aspect that stores the numeric return value of an expensive operation and that retrieves it on subsequent invocations. Different aspect instances (and thus cache values) are created for each `DataProvider` using the **perthis**`()` keyword. (In practice, the cache might employ a map to store a different return value per combination of argument values, but this was omitted for the sake of simplicity.) Until the cache has been initialized, the advice behaves as an augmentation advice that stores the original return value. Afterwards, it becomes a replacement advice that directly returns a value without executing the original behavior. Because the return value is both read and written in a single field, the same return type must be used in both signatures (rules *Body return value* and *Proceed use*). It is generally not safe to employ this advice for a method with a different return type than `Number` and rule *Advice return type* enforces this[6].

In previous research such as [15], it has been recognized that the implementation of a number of common design patterns benefit from the application of aspect-oriented programming. In listing 2,

[6] Although the entire cache could be made generic with respect to the type of data that it stores by introducing a type variable at the level of the aspect (as allowed in AspectJ 5 [9, sec. "Generic Aspects"]).

```
class Factory {
    Component createTextArea(String t) {
        return new JTextArea(t);
    }
}
aspect ScrollPaneFactory {
    pointcut componentCreation():
      execution(Component Factory.create*(..));
    JScrollPane around(): componentCreation():
      Component proceed() {
        return new JScrollPane(proceed());
    }
}
```

Listing 2. Factory Method Example

```
aspect Profiling {
    pointcut profile(): execution(* Main.*(..));
    <R> R around(): profile(): R proceed() {
        long start = System.currentTimeMillis();
        try { return proceed(); } finally {
            long stop = System.currentTimeMillis();
            reportMeasurement(stop - start,
              thisJoinPointStaticPart);
        }
    }
    //...
}
```

Listing 3. Example Profiling Aspect

we provide an example of the Factory Method pattern [12]. The intent of this pattern is to create an interface for object creation that defers instantiations to its specializations. Our example defines a factory that creates GUI components (only one factory method is shown). An aspect specializes the factory methods to decorate the created components with scrollbars. In this case, the aspect performs *replacement* advice that returns a newly created component. We can employ different return types for the two signatures of the advice method (rules *Body return value* and *Proceed use*), and as such the advice can be applied for join points with a return type between `JScrollPane` and `Component` (rule *Advice return type*). In case the specialized factory would refine the existing `Component` instead of creating a new value (e.g. a border can be defined for an existing component using the `JComponent.setBorder` method), the advice would qualify as augmentation advice and typing using type variables becomes possible.

Profiling is another example of a crosscutting concern that is often implemented using aspects (e.g. [20, sec. 5.6.2]). Listing 3 presents a profiling aspect that measures the execution time of methods in the `Main` class. Methods are identified using AspectJ's reflective access to the join point that is being advised. As can be expected, this behavior classifies as pure augmentation advice. By typing the advice method using an unbounded type variable as return type (rule *Type use*), it can be bound to join points of any return type (rule *Advice return type*).

3.3 Incorporation in Framework Approaches

As an alternative to the StrongAspectJ language, we now present an incorporation of the proposed typing principles of section 3.1 in a framework-based AOP approach.

AOP frameworks employ only base language constructs to describe aspectual behavior, and typically offer limited static type-safety guarantees in comparison to language extensions such as AspectJ. The AOP Alliance specification [27], which is implemented

```
interface JoinPoint<O,I> {
    O proceed(I i);
}
interface Advice<Oa,Ia,Op,Ip> {
    Oa around(Ia i, JoinPoint<Op,Ip> jp);
}
interface GenAdvice<Ob,Ib> {
    <O extends Ob, I extends Ib>
    O around(I i, JoinPoint<O,I> jp);
}
interface Pointcut<O,I> {
    void bind(GenAdvice<? super O, ? super I> ga);
    void bind(Advice<? extends O, ? super I,
      ? super O, ? extends I> a);
}
```

Listing 4. AOP Framework Key Interfaces

by a number of approaches including Spring AOP [18], even rules out all static type checking as all argument and return types are of the general `Object` type in its interfaces. The annotation-based style of AspectJ [9] improves in this respect by implementing advice methods as regular Java methods with special pre-defined annotations. As such, advice methods can be typed with a concrete signature which is checked for compatibility with the bound pointcut by the aspect weaver at load-time. However, to implement the *proceed*-method, a general interface `ProceedingJoinPoint` with `Object` argument and return types is used, again reducing safety guarantees to mere dynamic type-checking.

In this section, we show how the typing rules from our proposal can be enforced by an AOP framework that employs a generics-aware base language such as Java 5. The key interfaces for such a framework are shown in listing 4. We augment the core types representing join points, advice and pointcuts with type variables that represent the type declarations of these entities. In the versions presented in the listing, we assume one exposed pointcut parameter (type variables starting with `I`) and one return value (type variables starting with `O`). (In contrast to StrongAspectJ, we will explicitly keep track of pointcut return types.) The interface `Advice` encodes the two signatures of a non-generic *around*-advice. `GenAdvice` represents an advice that is generic in both argument type and return type; it can possibly declare bounds for these type variables. Finally, `Pointcut` encodes the binding rules for each of these two advice types respective to the types declared for the pointcut.

To demonstrate how the typing rules are enforced, we revisit an example from section 3.2.2, where an advice is presented that declares **`void`** as return type and `Person` as argument type in both the advice and *proceed* signatures. We reproduce this advice by implementing the interface `Advice`, instantiated with the appropriate concrete types. As such, advice bodies equivalent to those of the StrongAspectJ version become possible[7]:

```
class ExampleAdvice implements
  Advice<Void,Person,Void,Person> {
    Void around(Person i,
      JoinPoint<Void,Person> jp) {
        jp.proceed(new Person());
        //...
    }
}
```

The binding rules are then checked based on the types declared for the pointcut, represented by the (possibly wildcard) type arguments of type `Pointcut`. An instance of this advice

[7] With exception that the advice body cannot drop off its end and should instead return **`null`**, the only inhabitant of type `java.lang.Void`.

can be passed as an argument to the `bind`-method invoked on a term of type `Pointcut<Void,Person>`, but not on one of type `Pointcut<Void,? extends Person>`. Notice that this conforms to the prescribed typing rules, as the second type arguments of these two types represent the parameter type ranges `Person–Person` and `Null–Person` respectively.

The presented interfaces allow to enforce all typing rules from our proposal at compile-time using only the base language compiler (i.e. any standard compliant Java 5 compiler). However, a number of caveats still apply. It must still be enforced that pointcut definitions correspond to their declared parameter and return types (this depends on the manner in which the pointcut language is integrated in the framework). Also, the wildcard types in our proposal are more general than the wildcard type arguments currently available in Java 5: it is not possible to represent a double bounded range such as `Employee–Person`.

4. StrongAJ: An implementation using abc

In order to provide an experimentation platform for our approach, we have realized a prototype implementation of the StrongAspectJ language. Instead of developing an extension of the standard AspectJ compiler (*ajc*) [8], we have opted to employ the alternative AspectBench compiler (*abc*) [2], since it promises easy and modular addition of new aspect language features through plugins, without the need to fork the current source tree of the compiler. *abc* is itself built on top of the extensible compiler framework Polyglot, and the bytecode optimization library Soot. Following *abc* conventions, our plugin is named StrongAJ.

The StrongAJ implementation initially follows the *best practice* procedure for extending the *abc* platform. This involves extending the Polyglot frontend to support the new (or changed) syntax elements (e.g., in our case, the pointcut type ranges and the new *proceed* signature specification), and additionally providing the new abstract syntax tree (AST) nodes to represent these language constructs. By making these new nodes reachable to the Polyglot AST visitors, ambiguities in the type nodes are automatically resolved. Another visitor will type check nodes against the information from the type context, so by installing the new *proceed*-signature in this context when the visitor enters the scope of an advice declaration, the advice body is checked against its dual signature.

Verifying the pointcut/advice bindings requires more effort, but in general we can locate the existing type checks and extend them to enforce the additional type relations (typically adding a lower bound check in addition to the existing upper bound check). Pointcut arguments are checked in the frontend, where we override *abc*'s `typeCheck` method for named pointcut nodes. Return types are checked while weaving, so we are required to transport the additional type information of the advice signatures to the backend through *abc*'s so-called *AspectInfo* classes. The matching of **`this`**, **`target`** and **`args`** is also implemented in the backend, where, depending on the corresponding static join point type, they either never match, always match, or construct a test residue. We similarly install the new matching behavior at this point, and provide it with the declared type range of the variable being bound.

Since the current *abc* version (1.2.1) provides no support for Java Generics (nor for the other features of the Java 5 release from 2004), we were unsure about the feasibility of an addition of type variables to the type system. Despite our initial reluctance, this proved to be quite straightforward: we have introduced a new class `VariableType` as a subclass of *abc*'s `ReferenceType` class from the Polyglot type system, equipped with a supertype link to the type variable's bound. This is sufficient to verify the typing using variable types in advice bodies. However, since the backend cannot handle these types (type variables are not supported in Java bytecode), we have introduced a new frontend visitor pass to erase type variables from the AST after type checking. The type variables are replaced by their respective bounds, similar to the erasure procedure from Java Generics [4].

Although the current version of the StrongAJ plugin still resides in a proof-of-concept status[8], it implements the complete StrongAspectJ proposal, as verified by a test suite of 62 static and 3 dynamic test cases. Implementing it using the *abc* framework was a very reasonable effort, which required a total of 54 classes/interfaces (or about 2500 LOC). Nevertheless, we observe that some features require changing (i.e. subclassing) a large number of classes spread over both frontend and backend, sometimes only to pass required information to the relevant places. Perhaps an aspect-oriented implementation of the compiler itself could help tackling the cross-cutting nature of these features.

5. Defining pointcut/advice bindings formally

In section 3 we have informally introduced our proposal for type-safe pointcut/advice bindings. In this section we present a formalization of its essentials by presenting an excerpt of a corresponding type system we have developed. (The full type system, evaluation rules and property proofs, including for safety of the type system, are available as the technical report [10]; the evaluation rules, term typing rules and proof sketches are presented in appendix A.) We then revisit some of the examples introduced previously in order to show that they are correctly handled.

5.1 Type system

Our type system has been built based on a formal framework introduced by Jagadeesan et al. [17] for the typing of aspects in the presence of type polymorphism.

Figure 4 shows an excerpt of our type system that concerns pointcut and advice declaration. These type rules formalize part of the informal rules given in figure 3: the last four rules given here, for example, formally define the conditions stated in rules *Advice parameter type* and *Advice return type* of figure 3.

The typing judgments are based on variable types (e.g., X, Q) and non-variable types (e.g., C, F, G). Furthermore, we make use of the notations introduced previously, in particular, subtyping $<:$, type ranges C–D (cf. section 3.1.2), and (possibly zipped) sequences of types and terms, e.g. for typed argument lists $\bar{Q}\ \bar{x}$.

Pointcuts Although we do not use explicit type signatures for *pointcuts*, we expect that a pointcut ϕ can be typed in terms of the variables it exposes ($\bar{z}$), the type ranges for these variable values $\bar{F}$–$\bar{G}$, and the type range D–E of the return value. (Note that our type system accommodates return types for pointcuts while they are not present in StrongAspectJ as discussed in section 3.2.1.)

The two rules WELL-FORMEDNESS-PC and CONSISTENCY-PC in figure 4, which are noted 'upside down' compared to ordinary type rules, represent the requirements for the matching and typing semantics of a pointcut language that is abstracted from the core calculus. Rule WELL-FORMEDNESS-PC requires that this typing produces well-formed types and an environment whose entries corresponding to the pointcut parameters are well-typed (here and further on the auxiliary typing judgment z_i unique in $\bar{z}$ means that all elements, here z_i, of a sequence are unique in that sequence, here $\bar{z}$). Rule CONSISTENCY-PC specifies that only join points are matched that conform to a pointcut type. To this end, an auxiliary function match returns, for a given method call M and pointcut ϕ, the exposed variables $\bar{y}$, values $\bar{K}$ and a *proceed* body M'. A match then conforms to the pointcut type if the exposed values have types that specialize the corresponding upper bounds of the pointcut arguments, and if the *proceed* term can

[8] Available at `http://ssel.vub.ac.be/strongaj/`

WELL-FORMEDNESS-PC

$$\frac{\vdash \phi : D\text{–}E(\bar{F}\text{–}\bar{G}\ \bar{z})}{\vdash D, E, \bar{F}, \bar{G} \qquad z_i \text{ unique in } \bar{z}}$$

CONSISTENCY-PC

$$\frac{\begin{array}{c}\vdash \mathsf{match}(M, \phi) = \bar{y}; \bar{K}; M' \\ \vdash \mathtt{pointcut}\ \phi : D\text{–}E(\bar{F}\text{–}\bar{G}\ \bar{z}) \qquad \vdash M : R\end{array}}{\begin{array}{c}\bar{y} = \bar{z} \qquad \vdash \bar{K} : \bar{G}' \qquad \vdash \bar{G}' <: \bar{G} \qquad \bullet; \bar{F}\ \bar{y} \vdash M' : R' \\ \vdash R' <: E \qquad \vdash D <: R\end{array}}$$

DEC-ADVICE

$$\frac{\begin{array}{c}\bar{X} \triangleleft \bar{C} \vdash \bar{C}, R, \bar{P}, S, \bar{Q} \\ \bar{X} \triangleleft \bar{C}; \bar{P}\ \bar{x}, S\ \mathtt{proceed}(\bar{Q}) \vdash M : R' \\ \bar{X} \triangleleft \bar{C} \vdash R' <: R \\ \vdash \mathtt{pointcut}\ \phi : D\text{–}E(\bar{F}\text{–}\bar{G}\ \bar{z}) \\ \forall i. \bar{X}, \bar{C}, \bar{P}, \bar{Q}, \bar{F}, \bar{G} \vdash P_i, Q_i \text{ valid} \\ \bar{X}, \bar{C}, S, R, D, E \vdash R, S \text{ valid}\end{array}}{\vdash <\bar{X} \triangleleft \bar{C}>\ \ R\ \mathtt{around}(\bar{P}\ \bar{x}) : \phi : S\ \mathtt{proceed}(\bar{Q})\ \ \{M\}}$$

VALID-NONVARARG

$$\frac{\begin{array}{c}\forall k. P_i \neq X_k \wedge Q_i \neq X_k \\ x_i = z_j \qquad \vdash G_j <: P_i \qquad \vdash Q_i <: F_j\end{array}}{\bar{X}, \bar{C}, \bar{P}, \bar{Q}, \bar{F}, \bar{G} \vdash P_i, Q_i \text{ valid}}$$

VALID-VARARG

$$\frac{\begin{array}{c}P_i = Q_i = X_k \\ x_i = z_j \qquad \vdash G_j <: C_k \\ X_k \text{ unique in } R\ \bar{P}, S\ \bar{Q}\end{array}}{\bar{X}, \bar{C}, \bar{P}, \bar{Q}, \bar{F}, \bar{G} \vdash P_i, Q_i \text{ valid}}$$

VALID-NONVARRETURN

$$\frac{\begin{array}{c}\forall k. R \neq X_k \wedge S \neq X_k \\ \vdash R <: D \qquad \vdash E <: S\end{array}}{\bar{X}, \bar{C}, S, R, D, E \vdash R, S \text{ valid}}$$

VALID-VARRETURN

$$\frac{\begin{array}{c}R = S = X_k \\ \vdash E <: C_k \\ X_k \text{ unique in } R\ \bar{P}, S\ \bar{Q}\end{array}}{\bar{X}, \bar{C}, S, R, D, E \vdash R, S \text{ valid}}$$

Figure 4. Typing rules (excerpt)

be typed with an environment that assigns the pointcut arguments their lower bounds. The resulting type generalizes the lower bound of the pointcut return type range, while the return type of the original method call specializes the upper bound of the pointcut return type range.

Advice Advice declarations are typed using the last five rules of figure 4. We only consider the *around*-advice since it can emulate the other advice kinds. Rule DEC-ADVICE essentially specifies that, when the advice body is typed with an environment that contains the declared `proceed` signature, its return type R' must be a subtype of the declared return type R of the advice. The last two premises respectively ensure the conditions on the types of arguments and return values of the *around* and *proceed* signatures. Each of the two premises can be established by two rules, one rule for non-variable types and one for variable types[9]. As explained in section 3.2.1, in these rules, argument and return values that are in corresponding positions in the *around* and *proceed* signature have to be both variable or both non-variable. Furthermore, if they are variable the must denote the same type variable.

[9] The resulting four rules are in fact a slightly restricted, but more readable, version of the single more general rule used in our type system, see the rule DEC-ADVICE in appendix A.

The rule for non-variable argument types VALID-NONVARARG mainly states that pointcut argument type range $\bar{F}$–$\bar{G}$ is contained in the argument type range established by the *proceed* and advice $\bar{Q}$–$\bar{P}$, thus requiring a contravariant relationship. The rule VALID-VARARG essentially states that the upper bound of the pointcut argument range must specialize the type variable (no variance relationship here).

Similarly, The rule for non-variable argument types VALID-NONVARRESULT mainly states that the pointcut result type range D–E is contained in the result type range established by the (combined) *proceed* and advice R–S, thus requiring a covariant relationship. The rule VALID-VARRESULT essentially states that the upper bound of the pointcut result range must specialize the type variable (once again no variance relationship here).

Type safety Our type system is type safe: proof sketches for corresponding type preservation and typing progress properties can be found in appendix A, more details are available in the companion report [10].

5.2 Examples revisited

In order to illustrate the above rules, let us reconsider some of the motivating examples we have presented previously. The advice at the end of section 3.2.2 that executes a base method using *proceed* as long as its integer value is smaller than 100 is correctly typed as expected: the return type of the advice and the return type of the *proceed*-method are both `N`, i.e., both types are variable and equal and thus satisfy rule VALID-VARARG. Furthermore, according to rule DECL-ADVICE, methods to which this advice is applied must return a value of a subtype of the upper bound of variable `N`, i.e., `Number`.

In the case of the Factory Method example shown in listing 2, the advice declaration ensures through rule VALID-NONVARRESULT that the advice of aspect `ScrollPaneFactory` may only be applied to calls returning values of type in the range between `JScrollPane`, the return type of the advice, and `Component`, the return type of the *proceed* method.

6. Related Work

Typing problems of aspect languages and type variance for advice has been considered in recent work on the foundations of AOP and, rather in an ad hoc manner, in the context of several concrete aspect languages. We now consider relevant work of these two groups.

Wand et al. [31] have presented for the first time some of AspectJ's type safety problems in a precise formal framework. They have presented AspectJ's typing policy for non-generic *around*-advice and *proceed* invocations and illustrated its lack of soundness. This work does not, however, investigate remedies to this problem.

Three recent publications propose formal calculi that support a type-safe form of pointcut/advice bindings in object-based settings. Clifton and Leavens [6] introduce an imperative core language that models context exposing pointcut primitives as well as *around*-advice capable of changing parameter bindings on *proceed*-invocations. The authors 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 advices (the language employs non-variable types only). This precludes a large number of useful advices admitted by our approach.

Jagadeesan et al. [17] extend Featherweight Generic Java [16] with an advice construct whose type may depend on explicitly-declared type variables. For example:

```
advice <R extends Number> R Ex(): exe(R Foo.*()) {
  return do_after(proceed());
}
```

This enables typing of advice similar to the generic advice proposed in this paper. They present two safe type 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 this joint signature (i.e. no type variance, although the type variables from the signatures can be instantiated for each join point). The typing of *replacement* advice such as the Factory Method example we have tackled is therefore restricted [17, p. 21]. Furthermore, it is unclear how the inlined pointcuts could be decoupled from the advice declaration, since the scope of type variables also extends over the pointcut expression. (In the above example, note that the type variable `R` is used in the pointcut to quantify over execution join points of methods with a return type that is a subtype of `Number`.) In contrast, we explicitly support abstraction of pointcuts through typing based on type ranges.

Ligatti et al. [21] consider a type system for minimal core aspect languages, among others, in the context of an object-based base language. Since their approach (i) only considers advice having the same type as the join point triggering it and (ii) does not include subtyping between objects, their results only contribute marginally to the problems we have considered.

Aspectual Caml [23] is an aspect-oriented extension of the functional programming language Objective Caml. It includes a pointcut/advice mechanism that integrates with the static type system of the language. Pointcuts select join points through name and argument patterns, but are typed with type variables whose bindings are inferred from the advices to which the pointcuts are bound. The pointcut will then only select join points that match its typing. While this also enables typing (and type inference) of generic advice behavior using type variables, this work does not address the influence of subtype polymorphism (neither structural nor nominal) on the safety guarantees and type inference algorithms (which are Hindley-Milner based). It is therefore unclear how their conclusions can be translated to an object-oriented setting.

AspectJ 5 [9] modifies the AspectJ language to support Java 5 generics. Firstly, this involves coping with the presence of generics in the base language: type patterns can match generic types and their instantiations, and generic members can be defined through inter-type declaration. Secondly, type variables can be declared for aspects, similar to generic classes. When these variables are employed as regular type annotations in the aspect definition, this allows more advanced typing of generic aspect entities, akin to the advantages of generic classes. Additionally, type variables can also be employed in the type patterns of pointcuts and declare statements, where they directly influence the semantics of the aspect. As such, a new class of generalizations (as partially proposed in [14]) is made possible. Although generic aspects can generalize functionality over different deployments, it is not possible to declare type variables for advice methods. It is as such not possible to generalize over different applications of an advice method in one deployment, as generic advice allows. Furthermore, AspectJ 5 does not address any of the type-safety problems outlined in this paper.

Lohmann et al. [22] study the combination of AOP and C++ templates in the context of the AspectC++ language. One dimension of this work focuses on the usage of generic code in aspects. The AspectC++ compiler realizes *generic advice*, in their sense, by transforming advice code into a template member function that is called from each instrumented join point. A specifically-generated class that encodes the type information of the join point is passed a template parameter for this call, and as such, each case can be type-checked by the underlying C++ compiler. The usage of templates provides an expressive form of compile-time metaprogramming (Turing-complete even), exploited in AspectC++ to provide even more advanced kinds of *generative* advice. The trade-off is however that less abstraction is possible as type-checking of templates can only be done after their expansion. AspectC++ is therefore only capable of type-checking advices for a concrete join point at hand, while AspectJ (and our expansion of it) allows to type-check advices against declared pointcut parameter types, irrespective of a base application.

Finally, some work has been done on concrete language design issues concerning polymorphism and advice, notably Ernst and Lorenz's work on aspectual polymorphism in the context of AspectJ [11]. They propose a notion of advice groups from which the most specific advice is selected at runtime based on a late binding mechanism. Expressing the corresponding variance using typed generic advice as we have proposed improves on this because of the support for static type checking and better integration with the base language. The authors also consider the relationship of aspect instantiation and reflective access to polymorphic advice, two issues which should also be explored in the context of our approach.

7. Conclusions and Future Work

This paper presents a novel type system to recover safety for pointcut and advice declarations. As typing mechanisms, we propose separate signatures for *proceed* and corresponding *around*-advices, signature ranges for pointcuts and type variables for generic advices. For these elements, we derive type relations to guarantee safe advice application, and we show how they can support various representative kinds of advice behavior. We present StrongAspectJ, an integration of this type system with the AspectJ language, and provide an implementation as a plugin for the AspectBench Compiler. We also show how the typing principles can be statically enforced in an AOP framework by a non-aspectual (but generics-aware) compiler. Finally, we have presented formal definitions of the proposed constructs and a corresponding type system along with a proof of a corresponding type-safety property.

This work paves the way for a number of improvements to be tackled as future work. The more expressive typing constructs of our proposal sometimes result in quite complicated syntax forms. Investigating how we can simplify this for the programmer, e.g. by adding syntactic sugar or by inferring certain type declarations, is work in progress. Furthermore, it might be interesting to explore the usage of a pointcut expression typing not as a means of quantification, but as an aid in the development and maintenance of pointcuts; current pointcut errors often lead to wrong or empty matchsets, and can be difficult to debug.

References

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

[2] P. Avgustinov, A. S. Christensen, L. Hendren, et al. abc: An extensible AspectJ compiler. In P. Tarr, editor, *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*, pages 87–98. ACM Press, Mar. 2005.

[3] G. Bracha and D. Griswold. Strongtalk: typechecking Smalltalk in a production environment. In *Proc. of OOPSLA '93*, pages 215–230. ACM Press, 1993.

[4] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. of OOPSLA '98*, pages 183–200. ACM Press, Oct. 1998.

[5] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, Dec. 1985.

[6] C. Clifton and G. T. Leavens. MiniMAO1: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63(3):321–374, 2006.

[7] A. Colyer. Implementing caching with AspectJ. Blog entry: `http://www.aspectprogrammer.org/blogs/adrian/`, June 2004.

[8] A. Colyer et al. The AspectJ development environment guide. Available at `http://www.eclipse.org/aspectj/doc/released/devguide/`, 2002.

[9] A. Colyer et al. The AspectJ 5 development kit developer's notebook. Available at `http://www.eclipse.org/aspectj/doc/released/adk15notebook/`, Dec. 2005.

[10] B. De Fraine, M. Südholt, and V. Jonckers. Formal semantics of flexible and safe pointcut/advice bindings. Technical Report SSEL 02/2007/a, Vrije Universiteit Brussel, Oct. 2007. Available at `http://ssel.vub.ac.be/files/formal07a.pdf`.

[11] E. Ernst and D. H. Lorenz. Aspects and polymorphism in AspectJ. In Akşit [1], pages 150–157.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[13] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley Professional, third edition, June 2005.

[14] S. Hanenberg and R. Unland. Parametric introductions. In Akşit [1], pages 80–89.

[15] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proc. of OOPSLA '02*, pages 161–173. ACM Press, 2002.

[16] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proc. of OOPSLA '99*, pages 132–146. ACM Press, Oct. 1999.

[17] R. Jagadeesan, A. Jeffrey, and J. Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 63(3):267–296, 2006.

[18] R. Johnson et al. Spring Java/J2EE Application Framework. Home page at `http://www.springframework.org/`, 2004.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[20] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[21] J. Ligatti, D. Walker, and S. Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming*, 63(3):240–266, 2006.

[22] D. Lohmann, G. Blaschke, and O. Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In *Proc. of GPCE'04*, volume 3286 of *Springer-Verlag Lecture Notes in Computer Science*, pages 55–74. Springer, Oct. 2004.

[23] H. Masuhara, H. Tatsuzawa, and A. Yonezawa. Aspectual Caml: an aspect-oriented functional language. In *Proc. of ICFP '05*, pages 320–330. ACM Press, 2005.

[24] M. Naftalin and P. Wadler. *Java Generics and Collections*. O'Reilly Media, Inc., Oct. 2006.

[25] Object Management Group. *Unified Modeling Language 2.0 Superstructure Specification*, Feb. 2005.

[26] K. Ostermann and M. Mezini. Conquering aspects with Caesar. In Akşit [1], pages 90–99.

[27] R. Pawlak, R. Johnson, A. Popovici, et al. AOP Alliance (Java/J2EE AOP standard) version 1.0. Home page at `http://aopalliance.sourceforge.net/`, Mar. 2004.

[28] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for interactions in aspect-oriented programs. In *Foundations of Software Engineering (FSE)*, pages 147–158. ACM, Oct. 2004.

[29] D. Suvée and W. Vanderperren. JAsCo: An aspect-oriented approach tailored for component based software development. In Akşit [1], pages 21–29.

[30] M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. M. Gafter. Adding wildcards to the Java programming language. In H. Haddad, A. Omicini, R. L. Wainwright, and L. M. Liebrock, editors, *Proc. of the 2004 ACM Symposium on Applied Computing (SAC)*, pages 1289–1296. ACM Press, 2004.

[31] M. Wand, G. Kiczales, and C. 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.

A. Type system essentials

In this appendix we present the essentials of our type system. The reader is referred to [10] for a self-contained presentation.

A.1 Definitions

The following definitions make use of two standard auxiliary judgments for field lookup ($\Delta \vdash \mathsf{fields}(T) = \bar{P}\ \bar{f}$) and method lookup ($\Delta \vdash \mathsf{meth}(T.\ell) = {<}\bar{Y} \triangleleft \bar{E}{>}R(\bar{P}\ \bar{x})\{M\}$).

A.1.1 Evaluation

The evaluation rules of our calculus are shown in figure 5. (Besides these rules, the usual evaluation contexts for the congruent evaluation of subterms apply.) Method call evaluation first selects all declared advices (rule EVAL-SELECT), and reduces this list according to whether their pointcut matches (rule EVAL-APPLY) or not (rule EVAL-NOAPPLY). Only after all advices have been considered, the original method is executed (rule EVAL-METHOD). The substitutions in rule EVAL-APPLY are required to rename $\bar{y}$ into $\bar{x}$ in the pointcut arguments, for which an inner double substitution is employed. (Different typing rules 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 $\texttt{proceed}(\bar{N})$, where:

$$\texttt{proceed}(\bar{N})[{}^{M}/_{\texttt{proceed}(\bar{x})}] = M[{}^{\bar{N}}/_{\bar{x}}]$$

The outer substitution will bind any elements of $\bar{y}$ that do not appear in $\bar{x}$ back to their original argument value after proceed substitution.

A.1.2 Typing

The term typing rules are given in the left half of figure 6. These are equal to the rules of Featherweight Generic Java [16], but also include a rule to type a *proceed* invocation based on its signature in the term environment (rule TERM-PROCEED), and a rule to type an intermediate advised term (rule TERM-ADVISED). All rules assume additionally that the environment is well-formed (i.e. $\Delta; \Gamma \vdash \mathsf{ok}$).

The right half of figure 6 presents a generalized typing rule (DEC-ADVICE) for both generic and non-generic advice declarations. The conditions are similar to those presented in figure 4 except for the final binding rules. These basically state that, for any join point return type (represented by fresh variable W) and for any join point argument types (represented by fresh variables $\bar{Z}$),

$$\textsc{Eval-Field}\quad \frac{\vdash \mathsf{fields}(C) = \bar{f}}{\texttt{new}\ C(\bar{N}).f_i \rightarrow N_i}$$

$$\textsc{Eval-Method}\quad \frac{\begin{array}{c}\vdash \mathsf{meth}(C.\ell) = \texttt{<}\bar{X}\texttt{>}(\bar{x})\{L\}\\ M = \texttt{new}\ C(\cdots)\end{array}}{M.\ell\texttt{<}\bar{V}\texttt{>}[](\bar{N}) \rightarrow L[{}^{\bar{V}}/_{\bar{X}}, {}^{M}/_{\texttt{this}}, {}^{\bar{N}}/_{\bar{x}}]}$$

$$\textsc{Eval-Select}\quad \frac{\bar{b} = [a|\mathscr{D} \ni \texttt{advice}\ a\cdots{:}\cdots{:}\cdots]}{M.\ell\texttt{<}\bar{V}\texttt{>}(\bar{N}) \rightarrow M.\ell\texttt{<}\bar{V}\texttt{>}[\bar{b}](\bar{N})}$$

$$\textsc{Eval-NoApply}\quad \frac{\begin{array}{c}\mathscr{D} \ni \texttt{advice}\ a\cdots{:}\phi{:}\cdots\\ \vdash \mathsf{match}(M.\ell\texttt{<}\bar{V}\texttt{>}(\bar{N}), \phi) = \bot\end{array}}{M.\ell\texttt{<}\bar{V}\texttt{>}[a,\bar{b}](\bar{N}) \rightarrow M.\ell\texttt{<}\bar{V}\texttt{>}[\bar{b}](\bar{N})}$$

$$\textsc{Eval-Apply}\quad \frac{\begin{array}{c}\mathscr{D} \ni \texttt{advice}\ a(\bar{x}){:}\phi{:}\cdots\{L\}\\ \vdash \mathsf{match}(M.\ell\texttt{<}\bar{V}\texttt{>}(\bar{N}), \phi) = \bar{y}; \bar{K}; I.\hbar\texttt{<}\bar{U}\texttt{>}(\bar{J})\end{array}}{M.\ell\texttt{<}\bar{V}\texttt{>}[a,\bar{b}](\bar{N}) \rightarrow L[{}^{\bar{x}[{}^{\bar{K}}/_{\bar{y}}]}/_{\bar{x}}, {}^{I.\hbar\texttt{<}\bar{U}\texttt{>}[\bar{b}](\bar{J})}/_{\texttt{proceed}(\bar{x})}][{}^{\bar{K}}/_{\bar{y}}]}$$

Figure 5. Term evaluation rules ($M \rightarrow M'$)

$$\textsc{Term-Var}\quad \frac{\Gamma(x) = T}{\Delta; \Gamma \vdash x : T}$$

$$\textsc{Term-Field}\quad \frac{\begin{array}{c}\Delta \vdash \mathsf{fields}(T) = \bar{S}\ \bar{f}\\ \Delta; \Gamma \vdash M : T\end{array}}{\Delta; \Gamma \vdash M.f_i : S_i}$$

$$\textsc{Term-Advised}\quad \frac{\begin{array}{c}\Delta; \Gamma \vdash M.\ell\texttt{<}\bar{V}\texttt{>}(\bar{N}) : R\\ \mathscr{D} \ni \texttt{advice}\ \bar{a}\cdots\end{array}}{\Delta; \Gamma \vdash M.\ell\texttt{<}\bar{V}\texttt{>}[\bar{a}](\bar{N}) : R}$$

$$\textsc{Term-Method}\quad \frac{\begin{array}{c}\Delta \vdash \mathsf{meth}(T.\ell) = \texttt{<}\bar{Y} \triangleleft \bar{E}\texttt{>}R(\bar{P})\cdots\\ \Delta \vdash \bar{V} \qquad \Delta; \Gamma \vdash (M, \bar{N}) : (T, \bar{P}')\\ \Delta \vdash (\bar{V}, \bar{P}') <: (\bar{E}, \bar{P})[{}^{\bar{V}}/_{\bar{Y}}]\end{array}}{\Delta; \Gamma \vdash M.\ell\texttt{<}\bar{V}\texttt{>}(\bar{N}) : R}$$

$$\textsc{Term-Proceed}\quad \frac{\begin{array}{c}\Gamma(\texttt{proceed}) = S(\bar{Q})\\ \Delta; \Gamma \vdash \bar{N} : \bar{Q}'\\ \Delta \vdash \bar{Q}' <: \bar{Q}\end{array}}{\Delta; \Gamma \vdash \texttt{proceed}(\bar{N}) : S}$$

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

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

Figure 6. Term typing rules ($\Delta; \Gamma \vdash M : T$) and advice typing rule ($\vdash \texttt{advice}\ a\texttt{<}\bar{X} \triangleleft \bar{C}\texttt{>}R(\bar{P}\ \bar{x}){:}\phi{:}S(\bar{Q})\{M\}$)

there exists a valid binding $\bar{V}$ for the type variables $\bar{X}$, such that, under the assumptions that join point arguments and return type adhere to their pointcut bounds (respectively $\bar{F}$–$\bar{G}$ and D–E), the type relations with the advice signatures (i.e. $R(\bar{P})$ and $S(\bar{Q})$, after substituting $\bar{V}$ for $\bar{X}$) are honored. In the companion report [10], we prove that this rule is indeed more general than the rules from figure 4 by showing how each V_k can be constructed to satisfy these requirements.

The concrete pointcut expressions are abstracted in our framework, but we assume that its typing and matching will be consistent according to rule Consistency-PC from figure 4. Additionally, we stipulate that for a term with sufficiently evaluated subterms, a pointcut should either match or not match.

A.2 Properties

We will give proof sketches for preservation (subject-reduction) and progress; more detailed proofs are available in [10]. In what follows, we will assume that all top-level declarations are well-typed ($\forall i. \vdash \mathscr{D}_i$).

A.2.1 Preservation

Lemma 1 (Proceed-Substitutivity)**.** *Consider* $\Delta = \bar{Y} \triangleleft \bar{C}$ *and* $\bar{D}$ *such that* $\vdash \bar{D}$ *and* $\vdash \bar{D} <: \bar{C}[{}^{\bar{D}}/_{\bar{Y}}]$*. Additionally consider* $\Gamma = \bar{P}\ \bar{x}, S\ \texttt{proceed}(\bar{Q})$ *and* $\bar{N}$ *and* M *such that* $\vdash \bar{N} : \bar{P}'$ *with* $\Delta \vdash \bar{P}' <: \bar{P}$ *and* $\bullet; \bar{Q}\ \bar{z} \vdash M : S'$ *with* $\vdash S' <: S$*. It now holds that if* $\Delta; \Gamma \vdash L : T$ *then* $\vdash L[{}^{\bar{D}}/_{\bar{Y}}, {}^{\bar{N}}/_{\bar{x}}, {}^{M}/_{\texttt{proceed}(\bar{z})}] : T'$ *with* $\vdash T' <: T[{}^{\bar{D}}/_{\bar{Y}}]$*.*

The proof is carried out by an induction on the judgment typing L, making use of a similar type substitutivity lemma for basic types, subtypes and methods.

Theorem 2 (Preservation)**.** *If* $\vdash M : T$ *and* $M \rightarrow M'$ *then* $\vdash M' : T'$ *for some* T' *such that* $\vdash T' <: T$*.*

The proof proceeds by induction on each of the evaluation rules. In case of rule Eval-Apply, we have that the involved advice, pointcut, and intercepted method call are all well-typed. From the conditions of Dec-Advice and the conclusions of Consistency-PC, we can (by appropriately binding $\bar{Z}$) show that $\bar{Q}\ \bar{x}$ (i.e. the *proceed* argument declaration) is a stronger environment than $\bar{F}\ \bar{z}$ (i.e. the pointcut argument lower bound), and can (under the assumption that a stronger environment preserves term typing) be used to type the *proceed* term $I.\hbar\texttt{<}\bar{U}\texttt{>}(\bar{J})$. We can then apply the proceed substitutivity lemma to show (using other conditions of Dec-Advice and a binding of W) that the resulting term of the evaluation will indeed preserve the original type.

A.2.2 Progress

Theorem 3 (Progress)**.** *If* $\vdash M : T$ *then either* M *is a value or* $M \rightarrow M'$ *holds for some* M'*.*

For the proof, we consider each of the term typing rules that can establish $\vdash M : T$. The most interesting case is Term-Advised, where we distinguish between an empty and non-empty list of advices. In case there is still advice to be considered, we have required that either there is progress in a congruent evaluation context, or its pointcut either matches (in which case we can apply Eval-Apply) or not matches (in which case we can apply Eval-NoApply). In case there are no more advices, we have reached the actual execution of the method call. We can show this method call is well-typed, so this case is similar to the corresponding case in Featherweight Generic Java.