Call and Execution Semantics in AspectJ

Ohad Barzilay
School of Computer Science
Tel Aviv University
ohadbr@cs.tau.ac.il

Yishai A. Feldman
Efi Arazi School of Computer Science
The Interdisciplinary Center, Herzliya
yishai@idc.ac.il

Shmuel Tyszberowicz
Department of Computer Science
The Academic College of Tel Aviv Yaffo
tyshbe@mta.ac.il

Amiram Yehudai
School of Computer Science
Tel Aviv University
amiramy@post.tau.ac.il

ABSTRACT
The Aspect-Oriented Programming methodology provides a means of encapsulation of crosscuting concerns in software. AspectJ is a general-purpose aspect-oriented programming language that extends Java. This paper investigates the semantics of call and execution pointcuts in AspectJ, and their interaction with inheritance. We present a semantic model manifested by the current (1.1.1) release of AspectJ, point out its shortcomings, and present alternative models.

Categories and Subject Descriptors
D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics*

General Terms
Languages

Keywords
Aspect-oriented programming, AspectJ

1. INTRODUCTION
Many papers and books have been written about Aspect-Oriented Programming (AOP) in general, and about AspectJ in particular (e.g., [1, 2, 4]), as well as several papers giving formal semantics of simple aspect-oriented languages (e.g., [3, 5, 6, 8–10]), but none of them provides a precise (even if not completely formal) semantics of AspectJ. Such a semantics is necessary for language users to express their intent, and is crucial for tools that compile into AspectJ. For example, we are developing a design-by-contract [7] tool for Java. The main purpose of such a tool is to instrument the code to check assertions (method pre- and postconditions and class invariants) at run time. Existing tools we have examined perform this instrumentation in various ways, all of which have subtle errors. Our tool uses AspectJ instead of ad-hoc methods. While working on the tool, we discovered that some pointcuts we wrote did not yield the sets of join points that we expected. This has led us to conduct the study that we report on here.

We believe that a close examination of the semantics of AspectJ as manifested by the current implementation, and a discussion of the desired or "correct" semantics, is important to the AOP community. We hope that studies of the semantics of other parts of the language will follow. This paper investigates one of the subtle parts of AspectJ, namely, call and execution pointcuts and their interaction with inheritance. We present a semantic model manifested by the current (1.1.1) release of AspectJ, point out its shortcomings, and present alternative models. We note that Jagadeesan et al. [3] mention a few of these shortcomings, but do not discuss their deficiencies.

We follow the approach taken by authors of the AspectJ documentation and books by ignoring implementation issues. For the purpose of this paper, we are not interested in how code instrumentation is carried out, and in the practical constraints on which classes may or may not be instrumented. We similarly ignore the implementation of the matching between pointcuts and join points in AspectJ. Instead, we treat AspectJ as a black box, and examine its performance on carefully-chosen test cases.

2. CURRENT SEMANTICS OF ASPECTJ
The semantics of the wildcard operators ("`*`" and "`..`") inside call and execution pointcuts are easily specified by considering them to be an abbreviation for the (infinite) union of all possible expansions. We will therefore ignore wildcards in the sequel. Also, in order to simplify the presentation, we will deal only with void functions of no arguments. This will entail no loss of generality. Since static methods are not inherited, we will also ignore those in the sequel.

2.1 Call Semantics

Consider the pointcut specified by `call(void A1.f())`. This should capture all calls to the method `f` defined in class `A1`. Indeed it does, but that is due to the careful wording of the previous sentence. What happens if `f` is inherited from another class? In order to answer this question, we will consider the following hierarchy of classes:

```
public class A1
{
  public void f() {}
  public void g() {}
}

public class A2 extends A1
{
  public void h() {}
}

public class A3 extends A2
{
  public void f() {}
}
```

We then consider the following three variable definitions, in which the name of the variable indicates its static type and, if different, also its dynamic type:

```
A1 s1 = new A1();
A3 s3 = new A3();
A1 s1d3 = new A3();
```

It turns out that the pointcut `call(void A1.f())` captures the calls `s1.f()`, `s3.f()`, and `s1d3.f()`. Similarly, the pointcut `call(void A1.g())` captures the calls `s1.g()`, `s3.g()`, and `s1d3.g()`. It seems that even without the `+` subtype pattern modifier, which specifies subclasses, these pointcuts capture calls to the same method in subclasses, whether inherited or overridden. This may be a little surprising—what is the `+` modifier for, then?—but is consistent with the dynamic binding mechanism of Java. (We shall have more to say about the `+` modifier later.)

The pointcut `call(void A3.f())` captures the call `s3.f()` but not `s1d3.f()`. This implies that matching of call pointcuts is based on the static type of the variable, which is *not* consistent with the dynamic binding principle, but may perhaps be justified based on the information available at the calling point. However, the real surprise is that the pointcut `call(void A3.g())` does not capture *any* join points in our example, not even `s3.g()`! The only difference between `f` and `g` in `A3` is that `f` is overridden whereas `g` is only inherited. Thus, it seems that for matching to succeed, it is necessary for the method to be lexically defined within the specified class—inheritance is not enough. We use the term "lexically defined" to indicate that a definition (first or overriding) of the method appears inside the definition of the class.

Thus we are led to the following model. The semantics of a pointcut will be given as a set of join points, formalized as a predicate specifying which join points are captured by the pointcut. Consider the following definitions:

- a pointcut $pc_c = \texttt{call(void } C.f\texttt{())}$,
- a variable defined as $S\ x\ \texttt{= new}\ D\texttt{()}$, and
- a join point $jp = x.f\texttt{()}$.

That is, the pointcut specifies a class C, and the target of the join point has the static type S and the dynamic type D. (Obviously, D must be a descendant of S for this to compile correctly. We will denote this relationship by $S \subseteq D$.) Then:

$$jp \in pc_c \iff S \subseteq C \ \wedge\ f \text{ is lexically defined in } C.$$

2.2 Execution Semantics

Continuing with our example, we find that call and execution pointcuts capture exactly the same join points for `s1` and `s3` (we are ignoring other features of pointcuts, such as `this` and `target`). The only difference is in the treatment of `s1d3.f()`, which is captured by `execution(A1.f())` and `execution(A3.f())` but not by `call(A3.f())`. However, `execution(void A3.g())`, like the corresponding call pointcut, captures none of our join points. Thus, the rule for an execution pointcut

$$pc_e = \texttt{execution(void } C.f\texttt{())}$$

seems to be:

$$jp \in pc_e \iff D \subseteq C \ \wedge\ f \text{ is lexically defined in } C.$$

That is, the static type is replaced by the dynamic type. Again, this can be justified by the different type information available at execution join points, but is nevertheless an inconsistency in the semantics.

2.3 Subtype Pattern Semantics

The semantics of a subtype pattern such as `call(A1+.f())` should naturally be equivalent to the union of all possible expansions where `A1` is replaced by any of its descendants. This is indeed the case in AspectJ. However, because of the surprising semantics described above, this has a subtle interpretation. If

$$pc_c^+ = \texttt{call(void } C\texttt{+.}f\texttt{())}$$

is a call pointcut using subtypes, the matching rule is:

$$jp \in pc_c^+ \iff S \subseteq C \ \wedge\ f \text{ is lexically defined in some } F \text{ s.t. } S \subseteq F \subseteq C.$$

In particular, the pointcut `call(A1+.h())` captures `s3.h()`, because `h` is defined in `A2`, but the same join point is *not* captured by `call(A3+.h())`, even though `A3` has this method. This violates our expectation that `call(A3+.h())` should be a subset of `call(A1+.h())` that is identical for all join points in classes under `A3`.

Similarly, for

$$pc_e^+ = \texttt{execution(void } C\texttt{+.}f\texttt{())},$$

the matching rule is:

$$jp \in pc_e^+ \iff D \subseteq C \ \wedge\ f \text{ is lexically defined in some } F \text{ s.t. } D \subseteq F \subseteq C.$$

Variable definition:	$S\ x$ `= new` D`()`
Join point:	$jp = x.f$`()`
Pointcuts:	$pc_c =$ `call(void` $C.f$`())`
	$pc_e =$ `execution(void` $C.f$`())`
	$pc_c^+ =$ `call(void` C`+.`f`())`
	$pc_e^+ =$ `execution(void` C`+.`f`())`

$$
\begin{aligned}
jp \in pc_c &\iff S \subseteq C \ \wedge\ f \text{ is lexically defined in } C \\
jp \in pc_e &\iff D \subseteq C \ \wedge\ f \text{ is lexically defined in } C \\
jp \in pc_c^+ &\iff S \subseteq C \ \wedge\ f \text{ is lexically defined in some } F \text{ s.t. } S \subseteq F \subseteq C \\
jp \in pc_e^+ &\iff D \subseteq C \ \wedge\ f \text{ is lexically defined in some } F \text{ s.t. } D \subseteq F \subseteq C
\end{aligned}
$$

Figure 1: Semantics of current (1.1.1) AspectJ implementation.

2.4 Summary

The current semantics of AspectJ is summarized in Figure 1. It satisfies some of our intuitive expectations but violates others. The points on which AspectJ is consistent with the intuitive semantics are:

- Pointcuts with wildcards are equivalent to the union of all possible expansions.
- Pointcuts with subtype patterns are equivalent to the union of all pointcuts with subtypes substituted for the given type.
- The semantics of execution pointcuts is based on the dynamic type of the target.

On the following points the semantics of AspectJ deviates from our intuition:

- The semantics of call pointcuts is different from that of execution pointcuts, and depends on the static type of the target.
- Call and execution pointcuts only capture join points for classes where the given method is lexically defined.
- As a result of this, the difference between pointcuts with or without subtype patterns is subtle and unintuitive.

It is arguable whether pointcuts without subtype patterns should capture join points in subclasses at all. On the one hand, an instance of a class is *ipso facto* considered to belong to all its superclasses; this is reflected in the syntactic restrictions on assignment and parameter passing, and in the semantics of the `instanceof` operator. On the other hand, the existence of the subtype pattern modifier seems to imply the intention that a pointcut that does not use it refer only to instances of the specified class.

We believe that the lexical restrictions shown in these semantic definitions were unintended; their removal would greatly simplify the semantics. Some evidence that this is not the intended semantics comes from the following quote from one of the AspectJ gurus [4, p. 79]: "The [`call(* Account.* (..))` pointcut] will pick up all the instance and static methods defined in the `Account` class *and all the parent classes in the inheritance hierarchy*" (emphasis added). This is not true in AspectJ, but is intuitively appealing.

Another interesting clue is the fact (pointed out to us by one of the anonymous reviewers) is that when the AspectJ compiler is invoked with the `-1.4` switch, the set of joinpoints defined by call pointcuts changes, and the restriction on the lexical definition of the method in the designated class is removed. Curiosly, the behavior of execution pointcuts does *not* change even with this switch.

3. ALTERNATIVE SEMANTICS

If the current AspectJ semantics is inappropriate, we should propose one or more alternatives. As mentioned above, such alternatives should not restrict methods to be lexically defined in the designated class. Two questions remain:

1. should subclasses be included when the subtype pattern modifier does not appear in the pointcut; and
2. should call and execution pointcuts capture different join points.

These issues lead to four possible definitions of the semantics (see Figure 2). In these definitions we use the term "f exists in C" to denote the fact that the method f exists in class C, whether or not it is lexically defined (or overridden) in it. We use the term "broad" for those semantics that include subclasses even when subtypes are not indicated, and "narrow" for those that do not. The term "static" denotes semantics that use the static type for call pointcuts, and "dynamic" denotes those that use the dynamic type. It is important to note that although the join points captured by call and execution pointcuts are the same in the dynamic semantics, their properties (e.g., `this` and `target`) are different.

Each of the four semantics is consistent and reasonable. Perhaps the broad–dynamic semantics best reflects object-oriented principles, in that a reference to a class includes its subclasses, and the type that determines matching is the dynamic rather than static type of the variable. However,

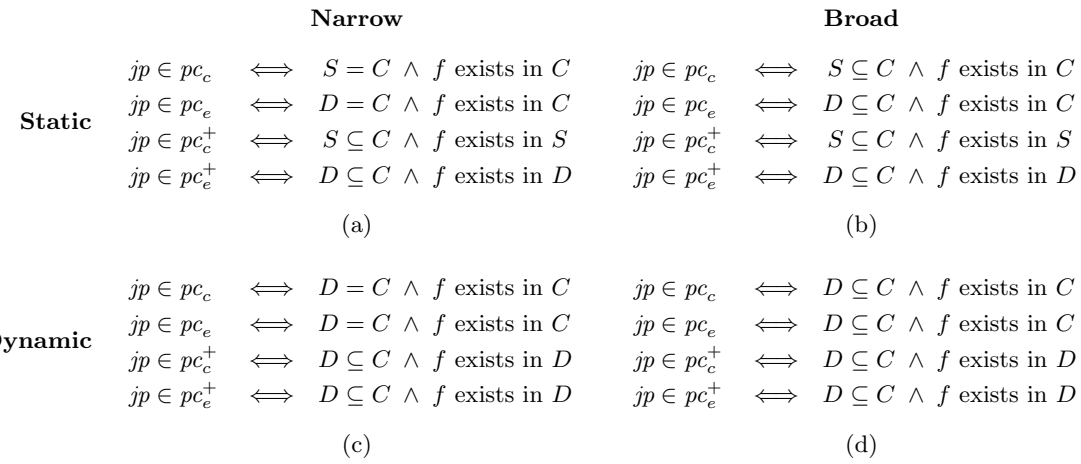

Figure 2: Four possible semantics: (a) narrow–static; (b) broad–static; (c) narrow–dynamic; (d) broad–dynamic.

other semantics may be easier to use if they more closely reflect the intent of AspectJ programmers.

4. EXPRESSIVE POWER

The five semantic models presented above (current AspectJ semantics and four alternatives) are able to describe different sets of join points. However, AspectJ has additional pointcut designators, which may be used to modify the meaning of a pointcut. The question now is, what is the expressive power of each of the given semantics definitions? Are there meaningful sets of join points that can only be expressed by some of them?

The answer is, of course, positive. For example, a narrow semantics is easily expressed in the corresponding broad semantics. The pointcut `call(void C.f())`, whose meaning in the narrow–static semantics is "$S = C \wedge f$ exists in C" can be expressed in the broad–static semantics by the following pointcut:

```
call(void C.f()) && target(x) &&
   if(x.getClass() == C.class)
```

However, the reverse is not true: in order to get a subset relation in the narrow semantics, we must use the subtype pattern modifier, but then there is no way to enforce the requirement that the method already exists in class C. So each broad semantics is strictly more expressive than the corresponding narrow semantics.

The static and dynamic semantics are incomparable. The dynamic semantics have no way of referring to the static type (S), and the static semantics have no way of referring to the dynamic type (D) in call pointcuts.

The semantics of `execution(void C+.f())` in either of the dynamic semantics is easily expressed in the current AspectJ semantics by the expression

```
execution(void f()) && this(C).
```

The corresponding expression for `call(void C+.f())` is

```
call(void f()) && target(C).
```

(Note that `target` in call pointcuts corresponds to `this` in execution pointcuts.) In order to understand the semantics of this expression under AspectJ, note that the call pointcut `call(void f())` without a class designator is equivalent to `call(Object+.f())`, so when applying the semantics of Figure 1, the class inclusion condition is trivial, and we obtain simply that f exists in S. Together with the additional requirement, `target(C)`, we get that the semantics of the above expression in AspectJ is

$$D \subseteq C \wedge f \text{ exists in } S,$$

which is a little different from the dynamic semantics.

Under the current semantics, AspectJ has no way of requiring that f exist in C without being lexically defined in it. The alternative, going to the top of the inheritance hierarchy, then prevents the possibility of referring to the static type. On the other hand, the new proposed semantics have no way of requiring the lexical definition of a method in some class. (Note that the `within` and `withincode` constructs are too restrictive, because they do not capture overriding definitions. Also, these do not help with call pointcuts, because they refer to the caller code rather than the method implementation.)

Of course, the fact that one semantics is more expressive than another does not mean it is better. The question is what programmers (and automatic tools) really need to say. Furthermore, the cost of a complex semantics should be weighed against the convenience of the language. Common patterns of usage should be expressed concisely. It might be better to adopt a simpler semantics for call and execution pointcuts, and add another construct to capture lexical definitions, if this is indeed necessary.

5. CONCLUSIONS

The current semantics of AspectJ has some unintuitive aspects. We have presented a number of alternative semantics, and compared their expressive power. The "right" semantics for AspectJ needs to be worked out with the user community, since it ultimately depends on how AspectJ is used in

practice. We hope that this paper will start a fruitful and constructive discussion on this question.

6. REFERENCES

[1] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Comm. ACM*, 44(10):29–32, 2001.

[2] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley & Sons, 2003.

[3] R. Jagadeesan, A. Jeffrey, and J. Reily. A calculus of untyped aspect-oriented programs. In Luca Cardelli, editor, *ECOOP 2003, European Conference on Object-Oriented Programming, Darmstadt, Germany*, volume 2743 of *Lecture Notes in Computer Science*, pages 54–73. Springer-Verlag, NY, 2003.

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

[5] R. Lämmel. A semantical approach to method-call interception. In *Proc. First Int'l Conf. Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, April 2002.

[6] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In *Ninth Int'l Workshop on Foundations of Object-Oriented Languages*, 2002.

[7] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[8] D. B. Tucker and S. Krishnamurthi. A semantics for pointcuts and advice in higher-order languages. Technical Report CS-02-13, Brown University, 2003.

[9] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*, pages 127–139. ACM Press, August 2003.

[10] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *Ninth Int'l Workshop on Foundations of Object-Oriented Languages*, 2002.