AOP and the Antinomy of the Liar

Florian Forster
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
Universitätsstraße 1
58097 Hagen, Germany
florian.forster@fernuni-hagen.de

Friedrich Steimann
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
Universitätsstraße 1
58097 Hagen, Germany
steimann@acm.org

ABSTRACT

Unless explicitly prevented, aspects can apply to themselves and can therefore change their own behaviour. This self-adaptation can lead to syntactically correct programs that express antinomies, i.e., that are meaningless (have no intuitive semantics). Drawing the parallel to mathematical logic, we suggest adopting the classical solution presented by Russell and Tarski, i.e., the separation of language into different levels. We propose a simple static type system for AOP that is based on such stratification and that not only helps avoid certain common programming errors, but also reflects on its inherent nature.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory – *Semantics, Syntax*

D.3.3 [**Programming Languages**]: Language Constructs and Features – *Recursion*

General Terms

Languages, Theory, Verification.

Keywords

Aspects, aspect-oriented programming, meta-programming, self-referentiality, antinomy, paradox, types

1. INTRODUCTION

AOP [4] [11] evolved out of meta programming [12]. It packs intercession, i.e., the possibility to intercept certain events in the course of a program and to insert event-specific behaviour, into a new language construct, the aspect.

Aspects are extremely powerful. In fact, they are so powerful that most contemporary implementations restrict their expressive power through certain syntactical constraints. For instance, most AOPLs do not let aspects advise other aspects (or even themselves). AspectJ [1][10], which has a primitive pointcut `adviceexecution()` that covers all executions of advices, provides constructs such as `cflow(.)` and `within(.)` (or, rather, `!within(.)`) that can be used to prevent self-reference and hence infinite recursion. However, these restrictions and by-passes are usually ad hoc in nature and not argued for on conceptual grounds; in fact, the general approach of language development seems to be that AOPLs are evolved according to their users' needs, and problems are fixed once they are discovered.

In this paper, we take a more principled approach to restricting the expressive power of AOPLs by revisiting a famous series of problems in logic and drawing the analogy to AOP. For this, we briefly recapitulate an ancient paradox known as the antinomy of the liar, and present certain variations of it that can be transformed into aspect-oriented programs (Section 2). Following the reasoning of the logicians who first solved the problem, we argue that any formal language allowing the expression of such antinomies is unsound, and needs mending (Section 3). In Section 4 we present several technical variants of a surprisingly simple solution that not only avoids all paradoxes of the discussed kind, but also other unwanted recursion of aspect application that until today can only be avoided by explicitly introducing certain run-time checks. In the discussion we compare our approach to related work, and find that it sheds some light on the nature of AOP.

2. FAMOUS ANTINOMIES AND THEIR TRANSLATION TO AOP

One of the oldest and also best known antinomies is that of the liar: when Epimenides the Cretan said that all Cretans are liars, and everything else they said was in fact untrue[1], he begged the question whether he himself told the truth, or lied. While the antinomy in Epimenides' utterance depends on certain assumption concerning the meaning of words, the paradox in it its simplest reduction,

"This sentence is false."

is fairly obvious: if the sentence is true, then by its meaning it must be false, and if it is false, the opposite of its meaning must be true, i.e., it must be true, thereby contradicting the presupposition.

This antinomy, which could not be resolved for some 2,500 years, has many incarnations. For instance, consider the following two sentences which, each one for itself being easy to understand, form an unpleasant loop ([8], p. 21):

[1] quoted after Bertrand Russell [13]. The original statement of Epimenides does not appear to have been formulated to provoke a contradiction.

1. The following sentence is false.
2. The preceding sentence is true.

Interpreting the first sentence as true makes the second sentence false which, assuming a binary (Boolean) logic, would make the first sentence false, thus making the second sentence true. Interpreting the second sentence as true makes the first one true and thus makes itself, the second sentence, false, and so on. There is no way out of this.

2.1 Formulations in AOP

Translating the above two sentences to an AspectJ program is almost straightforward. All we have to do is to replace the truth values *true* and *false* with execution and non-execution, respectively. Sentence 1 then translates to

```
public aspect S1 {
  void around(): adviceexecution() && within(S2) {
  }
}
```

i.e., the advice of `S1` negates the execution of `S2`'s advice (because it contains no `proceed()`). Accordingly, sentence 2 translates to

```
public aspect S2 {
  void around(): adviceexecution() && within(S1) {
    proceed();
  }
}
```

i.e., the advice of `S2` confirms the execution of `S1`'s advice. The intuitive semantics of these two aspects would imply that whenever the advice of `S1` is to be executed, it does not get executed, because the `proceed()` in the advice of `S2` (which would commence its execution) is cancelled by `S1`. Now if one accepts that execution of `S1`'s advice is cancelled, the advice of `S2` (the `proceed()`) does not get cancelled (by `S1`), so that there is not reason why `S1` should not get executed in the first place.

Starting the loop with the advice of `S2`, the picture is not much different: before `S2`'s advice can get executed, that of `S1` is executed, which cancels the execution `S2`'s advice and with it, through cancellation of `proceed()`, also cancels the execution of `S1`'s advice.

The operational semantics of AspectJ (as implemented by its compiler) has a simple solution to this paradox: since it calls the advices of both aspects in alternating order *before* it does anything (i.e., call or not call `proceed()`), it never comes to the core of the problem, but rather causes a stack overflow.

One might argue that `S2` is really a non-aspect, since it does not do anything other than intercept an invocation of `S1`'s advice and then continue with it. In fact, the following reduced aspect `S` could be thought of as inlining `S2` in `S1`:

```
public aspect S {
  void around(): adviceexecution() && within(S) {
    // do something, but do not proceed
  }
}
```

It could be interpreted as the programmatic form of "This sentence is false". Its intuitive semantics again would imply that whenever the advice of `S` is to be executed, it does not get executed, because the `proceed()` in the advice of `S` is lacking. Without a non-executed `proceed()`, however, there is no reason why `S` should not get executed. Admittedly, this is taking intuition a little far, but on the other hand, what is aspect `S` to express? Should it "do something", do nothing, or recur infinitely?[2]

Finally, the antinomy of the liar can be paraphrased in programming terms beginning with "all routines returning a truth value are always (i.e., for all calls) wrong". The passionate AO programmer might believe that this could easily be corrected by introducing a repair aspect, namely by

```
aspect Negate {
  Object around(): execution(* *(..))
      || adviceexecution() {
    Object c = proceed();
     if (c instanceof Boolean && c!= null)
       return !((Boolean) c);
     else
       return c;
   }
}
```

However, since the aspect would also have to correct itself, it is unclear what it should return in this case: upon execution, the above AspectJ code does the best it can — it runs into an infinite recursion, thus refusing to give an answer to the question.

2.2 Antinomies That Currently Cannot Be Expressed

There are also variations of the antinomy that cannot be expressed in AspectJ. Among the most famous is the barber who shaves all and only the people who do not shave themselves: assuming the barber shaved himself he would disregard the condition to shave only the people who do not shave themselves; assuming that he did not shave himself on the other hand he would, by the premise of his job description, have to shave himself. One way or another, the barber fails to meet the requirements of his task.

At first glance, this antinomy can be easily transcribed to AOP, namely to the following, informally defined aspect:

"Aspect `Barber` advises all and only the aspects
that do not advise themselves."

In AspectJ, that a concrete aspect `A` advises itself, i.e. its own pieces of advice, is expressed by the following pointcut:

`adviceexecution() && within(A)`

Conversely, that the aspect `A` does not advise itself is expressed by

`adviceexecution() && !within(A)`

The difficulty comes from generically expressing *all* aspects that advise, or do not advise, themselves. Due to existing language restrictions, AspectJ currently has no means of checking if an aspect advises itself. Whether intentional or not, this restriction saves AspectJ from being able to express the Barber's antinomy.

2.3 Non-Paradoxical Recursion

That the `adviceexecution()` pointcut designator can lead to infinite recursion is a well-known problem. In fact, in [6] it is stressed that

[2] As it turns out, it will recur infinitely as the advice is executed ("called") even though it does nothing. An optimizing aspect compiler might however change this semantics.

[t]he preferred way to use the `adviceexecution()` *pointcut is to pair it with* `within(YourAspect)`*, thus limiting its scope to advice appearing in the body of* `YourAspect`.

"The AspectJ Programming Guide" [3] gives a concrete example of this and shows how to avoid it:

```
aspect TraceStuff {
  pointcut myAdvice(): adviceexecution() &&
    within(TraceStuff);
  before(): call(* *(..)) && !cflow(myAdvice()) {
    // do something matching call(* *(..))
  }
}
```

However, the recursion that would occur in the application of `TraceStuff` if `!cflow(myAdvice())` were not included in the pointcut of the before advice can be considered a plain programming error.[3] In particular, it does not give rise to antinomies of the above kind, and its interpretation by the AspectJ compiler is not at conflict with its intuitive semantics. On the other hand, it is a programming error that is easily overlooked, one that would be nice if the language definition prevented the programmer from. We will return to this issue in Section 4.

2.4 Aspect Recursion Not Involving the Advising of Advice

Finally, we point the reader to the fact that there is a form of (usually unintended, i.e., erroneous) recursion that is caused by aspect application, but that does not involve the advising of aspects. The following gives an example of this:

```
public class Innocent {
   public void someMethod() {
     ...
   }
}

public aspect Naughty {
   before(Innocent a):
       execution(void Innocent.someMethod())
       && target(a) {
     a.someMethod();
   }
}
```

Note that recursion does not involve an `adviceexecution()` pointcut.

This kind of problem occurs when aspects access elements of the base program, thereby triggering (other) aspects including themselves. This however is of a different quality than the problems induced by the self-referentiality of aspects discussed above, and we make no proposals suggesting how to avoid such problems.

3. GREAT ESCAPES

It was one of the most significant mathematical discoveries of the early 20th century that antinomies of the presented kind are not the result of some linguistic sophistry, but rather question the fundamentals of all mathematical reasoning. In fact, mathematicians of that time (including Russell) seriously considered abandoning set theory altogether (and with it the concept of classes and relationships). Luckily for us, they did not, but instead came up with several solutions that avoided these problems. One of the earliest was formulated by Russell himself as his "theory of types", the essential idea of which, the distinction of different levels of propositions, was later repeated in Tarski's contemplations regarding the notion of truth. As it turns out, Russell's and Tarski's solution makes a useful contribution to AOP, but before transferring it to our problem, we briefly revisit the original works, one by one.

[3] In fact, in [2] the authors note that "circular `adviceexecution()` applications are very rare, and usually pathological and a symptom of an error in the program."

3.1 Russell's Theory of Types

In the year 1901 Russell discovered a fundamental problem in the naïve form of set theory that at that time was thought to be the basis of mathematics. In [13] he formulated "the class of all those classes which are not members of themselves":

$$M = \{X \mid X \notin X\}$$

The problem with this definition is that whichever of the two possible alternatives $M \in M$ and $M \notin M$ one assumes, the opposite follows:

$$M \in M \Rightarrow M \notin M$$
$$M \notin M \Rightarrow M \in M$$

Russell noted that the problem can only be avoided by agreeing that "[w]hatever involves *all* of a collection must not be one of the collection". However, the problem is that it is unobvious how to specify such a condition, since

[w]e cannot say: "When I speak of all propositions, I mean all except those in which 'all propositions' are mentioned"; for in this explanation we have mentioned the propositions in which all propositions are mentioned, which we cannot do significantly. [...] The exclusion [therefore] must result naturally and inevitably from our positive doctrines, which must make it plain that "all propositions" and "all properties" are meaningless phrases. [13]

Russell solved this problem constructively by introducing a "hierarchy of types":

A type *is defined as the range of significance of a propositional function, that is, as the collection of arguments for which the said function has values. Whenever an apparent variable occurs in a proposition, the range of values of the apparent variable is a type, the type being fixed by the function of which "all values" are concerned. The division of objects into types is necessitated by the reflexive fallacies which otherwise arise. These fallacies, as we saw, are to be avoided by what may be called the "vicious-circle principle", that is, "no totality can contain members defined in terms of itself". This principle, in our technical language, becomes: "Whatever contains an apparent variable must not be a possible value of that variable". Thus whatever contains an apparent variable must be of a different type from the possible values of that variable; we will say that it is of a* higher *type. Thus the apparent variables contained in an expression are what determines its type. This is the guiding principle in what follows.* [13]

Transferred to our problem of self-reference in AOP, the function

$$advice(joinpoint)$$

defines as a type the set of possible values the variable *joinpoint* may adopt. The value of *advice*(*joinpoint*) however must be of a higher type, so that it cannot be a value of *joinpoint*. It follows that no advice can serve as its own join point or, phrased differently, that no advice can advise itself. We will exploit this in our typing system for AOP described in Section 4.

It is interesting to note that Russell's type theory was only later generalized into sorted (and also order-sorted) predicate logic, whose sorts map closely to the types we know from typed programming languages. Since logic is usually restricted to first order, its sorts are all of Russell's type 1, i.e., they are sets of individuals (the objects). Our type system suggested in Section 4 lifts this restriction.

3.2 Tarkski's Distinction between Object Language and Meta-Language

In his discussion of the semantic conception of truth [17] Tarski analyzed the assumptions which lead to the antinomy of the liar, and observed the following:

> I. *We have implicitly assumed that the language in which the antinomy is constructed contains, in addition to its expressions, also the names of these expressions, as well as semantic terms such as the term "true" referring to sentences of this language; we have also assumed that all sentences which determine the adequate usage of this term can be asserted in the language. A language with these properties will be called "semantically closed."*
>
> II. *We have assumed that in this language the ordinary laws of logic hold.*
>
> *[...] Since every language which satisfies both of these assumptions is inconsistent, we must reject at least one of them.* [17]

Because the ordinary laws of logic are hard to renounce, it seems that semantic closedness cannot be upheld. Now if we agree

> *not to employ semantically closed languages, we have to use two different languages in discussing the problem of the definition of truth and, more generally, any problems in the field of semantics. The first of these languages is the language which is "talked about" and which is the subject matter of the whole discussion; the definition of truth which we are seeking applies to the sentences of this language. The second is the language in which we "talk about" the first language, and in terms of which we wish, in particular, to construct the definition of truth for the first language. We shall refer to the first language as "the object language," and to the second as "the meta-language."* [17]

Tarski further argues that in order to make statements about statements formulated in the object language, "the meta-language must be rich enough to provide possibilities of constructing a name for every sentence of the object language." Regarding truth, the meta-language must also contain terms of general logic such as AND, OR and NOT.

It springs to mind that the meta-language of Tarski and aspect languages (AspectJ in particular) have a lot in common. Quite obviously, since AspectJ extends Java, every sentence of the object language (Java) can occur in the meta-language (AspectJ). Names for expressions in the object language can be constructed by using pointcuts (the fact that it is not possible to construct a pointcut for every element of the object language is merely a limitation of the implementation). Last but not least, the meta-language contains logical terms for the formulation of pointcuts. Because we were able to reconstruct the antinomies in AspectJ, we conclude that it is semantically closed; in order to avoid them, we have to introduce a clear distinction between object language and meta-language.

4. TYPE-SAFE AOP

We will start the presentation of our solution with a practical example. It contains a recursion analogous to those presented in Section 2.3 and [3], but no antinomy. However, as we will elaborate later our solution is powerful enough to also avoid all antinomies we were able to express in Section 2.1, as well as ones that cannot (yet) be formulated, enabling certain future language extensions that seem too risky today.

One of the best known (and most often cited) applications of aspects is tracing: if the execution paths of a program become unobvious, a trace may help to find out what exactly is going on. However, because of its obliviousness property AOP comes with its very own tracing demands: the programmer might be particularly interested when a certain aspect (or all aspects) are executed or, more challenging, in which order certain conflicting pieces of advice are executed on the same join point[4].

Writing an advice that traces all method executions and advice executions seems an easy exercise. The first solution a programmer might propose, namely

```
public aspect Tracing {
  void around(): adviceexecution()
    || execution[5] (* *(..)) {
    System.out.println("Entering:" +
      thisJoinPoint);
    proceed();
    System.out.println("Leaving: " +
      thisJoinPoint);
  }
}
```

as a tracing aspect that traces both method and advice executions, as for instance

```
public aspect Worker {
  void around(): execution(* *(..)) {...}
}
```

and

```
public class Base {
  public void doSomething() {...}
}
```

does not work. Taking a closer look reveals that the pointcut attached to the tracing advice selects the tracing advice itself (by means of the unrestricted primitive pointcut `adviceexecution()`), sending AspectJ into infinite recursion. This is clearly a programming error, which has to be fixed somehow.

---

[4] Note that by the current definition of advice precedence in AspectJ this order might be impossible to determine. Even worse, it may change in between two compiler runs. [5]

[5] Although we consistently use the pointcut designator `execution(.)` for referring to the base program throughout the following, it should be understood that it could be replaced by other pointcut designators such as `set(.)` or `get(.)`.

An immediate solution would appear to be using the pointcut designator `within(<TypePattern>)`, where `<TypePattern>` identifies a number of classes, interfaces and/or aspects. The formerly unrestricted pointcut `adviceexecution()` can then anded with (restricted by) `!within(Tracing)`, i.e. only pieces of advice which are not in the lexical scope of the aspect `Tracing` are selected, as the following example shows.

```
public aspect Tracing {
  void around(): (adviceexecution()
    && !within(Tracing))
    || execution (* *(..)) {
    ...
  }
}
```

As it turns out, however, this construction cannot avoid indirect recursion. In fact, when applying it to aspect `S1` from Section 2.1, it must remain ineffective, since `within(S2)` implies `!within(S1)`. Therefore, one has to check explicitly whether `S1` has already been activated, a test that can be performed with the aid of the `cflow()` function. Hence, in order to be sure that self application is under all circumstances avoided, one has to include the verbose construct presented in Section 2.3. Thus, our tracing aspect becomes the clumsy

```
public aspect Tracing {
  pointcut guard(): adviceexecution() &&
    within(Tracing);
  void around(): (adviceexecution()
    || execution (* *(..))) &&
    !cflow(guard()) {
    ...
  }
}
```

It seems that the introduction of `adviceexecution()` as a means to let aspects apply to aspects has made necessary a programming pattern that serves to fix the resulting problems. However, this pattern means that the infinite recursion introduced by `adviceexecution()` has to be *explicitly* detected and broken, and this *at runtime*. What would be desirable instead is that `adviceexecution()`, while allowing certain (wanted) recursion, can never mean the (generally nonsensical) infinite recursion to itself.[6] In the following, we build such a solution on a theory of types as proposed by Russell or, equivalently, on a theory of object language and meta-language as proposed by Tarski. We develop the solution in a stepwise manner, by first presenting a programming pattern using annotations to introduce type (or meta) levels, then sketching a preprocessor utility for AspectJ that frees the programmer from the coding overhead and error-proneness of the pattern, and finally by suggesting the addition of a new keyword `meta` to AspectJ whose semantics does the job all automatically.

4.1 Step 1: Using Annotations and a Simple Programming Pattern

The basic idea of the solutions of Tarski and Russell was the introduction of different levels of language. What we need, therefore, is a way to organize the elements of an aspect-oriented program into several levels. With the new annotation feature of Java 5.0 and AspectJ5 one can introduce such stratification, thereby simulating the distinction of Java as the object language and AspectJ as the meta-language, or the type levels of Russell.

We declare the annotation needed for this purpose as follows:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface TypeLevel {
   int value() default 0;
}
```

Note that our annotating `TypeLevel` with the built-in meta-annotation `@Retention(RetentionPolicy.SOURCE)` implies that we evaluate the annotations statically (in contrast to `cflow()`, which can only be evaluated dynamically!). A second built-in meta-annotation, `@Target`, is set to `ElementType.TYPE`; it prevents the annotation of elements other than types, i.e. classes, interfaces, and aspects, by prompting a corresponding compilation error.

Our `TypeLevel` annotation has one argument which represents the meta-level of the annotated element. By definition the elements (class or interface) of the object language will have a meta-level of 0, meaning that they must be annotated with `@TypeLevel(0)`, and the elements (aspects) of the meta-language addressing elements of the object language (i.e., advice without an `adivceexecution()` in its pointcut) will have a meta-level of 1, meaning that they must be annotated with `@TypeLevel(1)`.[7] Thus, the base of our aspect `Tracing` must be annotated as

```
@TypeLevel(0)
public class Base {
  public void doSomething() {...}
}

@TypeLevel(1)
public aspect Worker {
  void around(): execution(* *(..)) {...}
}
```

When moving to the next higher level, the (former) meta-language becomes the (new) object language, so that the (new) meta-language ranges at level 2: advice with an `adviceexecution()` in its pointcut is to be annotated with `@TypeLevel(2)` or higher:

```
@TypeLevel(2)
public aspect Tracing {
  void around(): adviceexecution() {...}
}
```

The problem that remains is how to restrict the scope of the `adviceexecution()` pointcut to aspects of levels lower than that of its enclosing aspect. As it turns out, AspectJ 5 is equipped with the `@within(Annotation)` pointcut designator that matches only join points belonging to a type annotated with `Annotation`. By adding `@within(TypeLevel)` plus the explicit guard `if(TypeLevel.value() < 2)` to the pointcut, our tracing aspect can be formulated as

```
@TypeLevel(2)
public aspect Tracing {
  void around(): (adviceexecution()
    && @within(TypeLevel)
    && if(TypeLevel.value() < 2))
    || execution (* *(..)) {
    System.out.println("Entering: " +
```

[6] Note that this cannot be achieved simply by excluding every occurrence of `adviceexecution()` from its own scope, since the recursion may be indirect. *Cf.* also Russell's comment on the impossibility of explicit avoidance of self-reference in Section 3.1.

[7] Note that both Russell and Tarski introduced no absolute, but only relative levels. However, since our domain is AOP, the level of the (non-aspect) base program is as low as we can get.

```
      thisJoinPoint);
    proceed();
    System.out.println("Leaving: " +
      thisJoinPoint);
  }
}
```

without limiting its meaning unduly.

Unfortunately, things are not as simple with the current implementation of AspectJ, as the following example shows:

```
@TypeLevel(2)
public aspect Tracing {
  void around(): (adviceexecution()
    && @within(TypeLevel)
    && if(TypeLevel.value() < 2))
    || execution (* *(..)) {
      helpMethod();
    }
  }
  void helpMethod() {...}
}

@TypeLevel(1)
public aspect Worker {
  void around(): execution(* *(..)) {...}
}
```

When including in the aspect `Tracing` an arbitrary helper method (here: `helpMethod()`) and calling it from the aspect's advice, aspect `Worker` (which advises *all* method executions) advises the execution of this method, and therefore indirectly also the aspect `Tracing` even though it is of a higher type level. This leads to an infinite recursion as the execution of `Worker`'s advice triggers the advice of `Tracing`'s which executes the method `helpMethod()` again. The only way out of this (without checking the call stack) is to exclude `execution()` from applying to methods defined within aspects.[8] To achieve this, `execution(.)` pointcuts also have to be guarded:

```
@TypeLevel(2)
public aspect Tracing {
  void around(): (adviceexecution()
    || execution (* *(..)))
    && @within(TypeLevel)
    && if(TypeLevel.value() < 2) {
    helpMethod()
  }
  void helpMethod() {...}
}

@TypeLevel(1)
public aspect Worker {
  void around(): execution(* *(..))
    && @within(TypeLevel)
    && if(TypeLevel.value() < 1) {
    ...
  }
}
```

The accidental recursion is thus removed. It follows immediately that annotating base type (classes and interfaces) with `@TypeLevel(0)` cannot be avoided, although at first glance this seems to be redundant (because the base has `execution` and other exclusive pointcut designators that cannot apply to pieces of advice, and because it cannot be caught by `adviceexecution()`).[9]

Unfortunately, this solution has several problems. First, it only works if all types are tagged with their corresponding annotation, because if a type (class, interface, or aspect) is not annotated, a guarded pointcut will not select its join points, voiding all its aspects. Second, the programmer is responsible for ensuring the constraint that the value of the type guard of a pointcut is always lower than its own aspect's type level (because there are no means to instruct the compiler to check annotation values). Last but not least, the required code is highly stereotypical (it is in fact a coding pattern), and experience teaches that the implied programming overhead will not be welcomed by practicing programmers, particularly if workarounds requiring less coding (the `within(.)/ !cflow(.)` pattern) are available. Since annotating types and guarding advice cannot be enforced by the compiler, it will most likely not be used. On the other hand, much of the task is so stereotypical that it can be delegated to a pre-processor, as discussed next.

4.2 Step 2: Using a New Built-in Annotation

The next major Java release (codenamed "Mustang") will allow user-defined annotations to be included into the compilation process by means of a special interface to the compiler [9]. Once available, this pre-processing facility should allow us to extend the compiler with a pre-processor reducing the work and responsibility of the developer, and thus the likelihood of making errors. In this section we will therefore sketch such a pre-processor which, in concert with a correctly annotated program, statically ensures that the typing conditions of our language are satisfied.

In our description, we assume a procedure for pre-processing described in the Annotation Preprocessing Tool Manual [16]. The pre-processor for the tagging task, making sure that every aspect is appropriately annotated, is straightforward to write:

```
foreach type in program
  if isTypeTagged(type)
    do nothing
  else
    if (type == Class || type == Interface)
      type.tagWithLevel(0)
    if (type == Aspect)
      type.tagWithLevel(1)
endfor
```

Therefore, when feeding an untagged program to the pre-processor, it assumes that it consists of only base program and level 1 aspects, but no aspects advising aspects.[10] After this pre-processing step every type is, either by the pre-processor or by the developer, tagged with a `TypeLevel` annotation.

In the next step the pre-processor must generate the guards which are required to complete stratification of our language. The following pseudo code attaches the code pattern `@within(TypeLevel) && if(TypeLevel.value() < t)`, where `t` is the type level of the enclosing aspect, to every pointcut (un-

[8] One may ask oneself why AspectJ, while granting advice execution a different status ("higher level") than method execution, does not extend this to the methods defined within the aspect, in particular since inlining these methods should not change the meaning of the aspect.

[9] Note that if one insists that `helpMethod()` in `Tracing` is of type level 0 (the AspectJ view; *cf.* Footnote 8), i.e., part of the base, the resulting recursion is analogous to that discussed in Section 2.4, meaning that it cannot be broken by our type system.

[10] As we will see below, occurrence of an `adviceexecution()` pointcut in such a program will flag an error.

named or named) in the lexical scope of the aspect under investigation.

```
foreach aspect in program
  foreach pointcut in aspect
    t := getTypeLevel(aspect)
    attachGuard(pointcut, t)
  endfor
endfor
```

The generated guard allows the pointcut to select only join points occurring in the lexical scope of types annotated with a type level below its own. Thus our tracing advice

```
@TypeLevel(2)
aspect Tracing {
  void around(): adviceexecution()
    || execution (* *(..)) {...}
}
```

will be automatically extended to

```
@TypeLevel(2)
aspect Tracing {
  void around(): (adviceexecution()
    || execution (* *(..)))
    && @within(TypeLevel)
    && if(TypeLevel.value() < 2) {...}
```

whereas

```
@TypeLevel(1)
aspect Tracing {
  void around(): adviceexecution()
    || execution (* *(..)) {...}
}
```

will be extended to

```
@TypeLevel(1)
aspect Tracing {
  void around(): (adviceexecution()
    || execution (* *(..)))
    && @within(TypeLevel)
    && if(TypeLevel.value() < 1) {...}
```

which has an empty pointcut, because `adviceexecution()` only matches to program elements in the scope of type level 1 or higher. At this point, the pre-processor should flag a type level mismatch error.

The annotation-based pre-processing suffers from one rather subtle problem: it assumes that all pointcuts are intended to refer to program elements of *any* lower level. However, a programmer might want to specify that `adviceexecution()` should match advice at a particular level (and no other), and this level need not even be precisely 1 below itself. In such a case, an explicit guard (involving "=" rather than "<") will be required. We will present a more elegant solution for this in the next step.

4.3 Step 3: Extending the Language with the `Meta` Modifier

Although the exploitation of "semantic" (i.e., built-in, but user-defined) annotations reduces the programming overhead and the possibility to make mistakes, it is still only a precursor to full language support. In particular, it would be desirable for the compiler to detect and flag all errors related to the typing of aspects, just as it discovers other, conventional typing errors. Also, we believe that our suggested typing of aspects deserves the status of a new, native language construct, since it addresses a fundamental problem inherent in AO languages. Therefore, we propose a small, yet very effective extension to AspectJ which equips it with a type system à la Russell (not to be confused with the type system of Java) and Tarski, allowing the safe advising of advice.

Frankly, in our extended language attempting to compile

```
public aspect Tracing {
  void around(): adviceexecution() ...
}
```

would result in an error message "type level mismatch error: consider modifying aspect Tracing with meta", because `adviceexecution()` may refer to itself. The keyword `meta` preceding an aspect definition lifts the so-declared aspect up one level, i.e., it declares it as an aspect of both aspects and base programs (where the former must themselves be aspects of base programs, not of aspects). For instance,

```
meta aspect Tracing {...}
```

makes `Tracing` a meta aspect that can apply to the base program and aspects (`Base` and `Worker` in the above example), but not meta aspects, thereby excluding self-reference. The pointcut definition of `Tracing` can remain as is; in particular, it need not be explicitly guarded: it can refer only to lower levels by the definition of the language.

Now the lifting procedure can be applied repeatedly, raising the meta level of aspects even further. Although there will most likely be no need for having aspects on a level higher than 3 (given the usual four-layer architecture), there seems to be no obvious theoretical bound to meta levels. Therefore, rather that introducing ever new `meta` modifiers, we propose to denote the meta-meta level with `meta^2`, and generally concatenation of n metas by `meta^n`. `meta` is then simply shorthand for `meta^1`, and absence of `meta` is shorthand for `meta^0`. However, it is important to note that theoretically, for $n > 0$ each `meta^n` represents a different keyword of our language, and our shorthand notation is only introduced to allow the compiler to accept them as they are used in a program. We will return to this subtlety in Section 5.4. Here, we note that an aspect that is to apply to `Tracing` would have to be declared as

```
meta^2 aspect GodAspect {...}
```

or higher.

Following Russell's theory of types, we allow meta-aspects to apply to aspects as well as to base programs, rather than to aspects alone. We have no particular reason for this other than that we do not want to place unnecessary restrictions on the formalism. Had we decided that aspects can exclusively apply to program elements one level below them, no distinction between the `execution()` and the `adviceexecution()` pointcut designator would have been necessary: `execution()` would have sufficed (denoting base code or aspect execution, depending on the level of the defining aspect).

With the possibility of `adviceexecution()` pointcuts spanning arbitrary levels, we may wish to have increased precision available for expressing specifically to which level a pointcut applies. For instance, while we can already distinguish between base program (`execution(.)`) and aspect (`adviceexecution()`) and thus between type level 0 and higher levels, we may wish to be able to differentiate in our pointcuts between type level 1 and 2. Therefore, we allow that the pointcut designator `adviceexecution()` can also be modified with the keyword `meta`, specifying the exact type level to which the so-modified pointcut is to apply. The pointcut

```
pointcut metaAdvice(): meta adviceexecution();
```

would thus select only advice defined in aspects of type (or meta) level 2, like our aspect `Tracing` from above. The meaning of the (unmodified) pointcut designator `adviceexecution()` is then restricted to aspects of type level 1, i.e., those that are not meta aspects. Our tracing aspect can thus be rewritten as

```
public meta aspect Tracing {
  void around(): adviceexecution()
    || execution (* *(..)) {
    ...
  }
}

public aspect Worker {
  void around(): execution(* *(..)) {...}
}

public class Base {
  public void doSomething() {...}
}
```

Note that, as mentioned at the beginning of this subsection, using the pointcut designator `adviceexecution()` in an ordinary (i.e., non-meta) aspect or, generally, `meta[^n] adviceexecution()` in any aspect declared as `meta[^m]` with $m \le n$, would result in a compilation error, since it violates the typing rules of our language extension. The following table summarizes what is possible.

Aspect level	allowed pointcut designators
`aspect`	`execution()`
`meta aspect`	`execution(), adviceexecution()`
`meta^2 aspect`	`execution(),adviceexecution(), meta adviceexecution()`
`meta^3 aspect`	`execution(),adviceexecution(), meta adviceexecution(), meta^2 adviceexecution()`
…	…

This so modified AspectJ is now type safe in terms of the type theory of Russell, and the antinomies presented in Section 2 cannot be formulated in it, as the following demonstrates.

4.4 Resolving the Antinomies

With our new type system implemented, the code translation of the two contradictory sentences from Section 2.1 would now result in a type level mismatch (compilation) error, because the included (indirect) self-reference, i.e. `adviceexecution()`, while applying to type level 1, is in the lexical scope of an aspect of type level 1. In order to be well-typed, both `S1` and `S2` must be modified with `meta` as in

```
public meta aspect S1 {
  void around: adviceexecution() && within(S2) {
  }
}

public meta aspect S2 {
  void around: adviceexecution() && within(S1) {
    proceed();
  }
}
```

This however automatically prevents the self-reference and thus the infinite recursion. In fact, both pointcuts do not select any join point, since `adviceexecution()` implicitly applies to type level 1 whereas `within(S2)` and `within(S1)` apply to type level 2, so that the conjunction is always false. A corresponding compiler-generated error, or at least a warning, to notify the developer of this problem would seem desirable.

In the same vein, the recursions of all other paradoxical aspects presented above are naturally resolved. For instance, by modifying the declaration of the `Negate` aspect to `meta aspect Negate` eliminates the possible self-reference, and thus the need to restrict `adviceexecution()` by means of other pointcuts like `within(.)` and `cflow(.)`. The same holds for the unwanted recursion warned of in Refs. [3] and [6].

4.5 Handling of Aspect Members and Inter-Type Declarations

In Section 4.1 we mentioned that the weaving of AspectJ treats methods extracted from an advice as ordinary (base) methods, and showed how this can lead to indirect recursion. To solve this problem in our proposed extension of AspectJ, we assign to every join point in an aspect the type level of that aspect (cf. Footnote 8). Therefore, it cannot be matched by pointcuts of the same or lower levels, breaking the recursion.

To allow selective matching of the join points of an aspect exposed by its members (methods and fields), we further extend AspectJ to allow modification of all other pointcut designators (i.e., `call(.)`, `execution(.)`, `set(.)`, `get(.)`, etc.) with `meta^n`. `meta^n <pointcut>` will select only join points occurring in the lexical scope of aspects of the corresponding level. Using `meta^n <pointcut>` in an aspect declared as `meta^m` with $m < n$ will result in a (statically discovered) type level mismatch error. The complete table of admissible pointcut designators in each aspect type level is the following:

Aspect level	allowed pointcut designators
`aspect`	current AspectJ pointcut designators excluding `adviceexecution()`
`meta aspect`	same as above plus `adviceexecution()` plus every other AspectJ pointcut designator modified with `meta`
`meta^2 aspect`	same as above plus `meta adviceexecution()` plus every other AspectJ pointcut designator modified with `meta^2`
…	…

In order to catch all method executions in the base program and its (type level 1) aspects, our tracing aspect has to be rewritten as

```
public meta aspect Tracing {
  void around(): adviceexecution()
    || meta execution (* *(..))
    || execution (* *(..)) {
    ...
  }
}
```

It traces both the advice and the (helper) method of

```
public aspect Worker {
  void around(): execution(* *(..)) {
    someMethod();
  }
  void someMethod() {...}
}
```

For the convenience of the programmer it might prove useful to allow modification through `meta^n` also for defining the scope of named pointcuts. Instead of writing

```
void around(): meta get(* *)
  && meta set(* *) {...}
```

one could then write

```
meta pointcut accessors(): set(* *) && get(* *);
void around(): accessors() {...}
```

One remaining issue is that of how AspectJ's inter-type declarations are to be integrated into our typing system. Returning to our tracing example once more, we extend the aspect `Tracing` with an introduction affecting the aspect `Worker`.

```
public meta aspect Tracing {
  void around(): adviceexecution()
    || meta execution (* *(..))
    || execution (* *(..)) {
    ...
  }
  void Worker.doGood() {...}
}

public aspect Worker {
  void around(): execution(* *(..)) {...}
}

public class Base {
  public void doSomething() {...}
}
```

According to the current semantics of AspectJ the introduction `Worker.doGood()` is considered to be a member of the type it is introduced to, i.e., at least as regards join point matching it is equivalent to defining the method in the aspect `Worker` directly. This is in accord with our typing system: any aspect introducing elements to lower level types can also watch over their execution. For instance, in the above example the tracing aspect traces all executions of `doGood()` in `Worker`. Currently, we can see no need to restrict introductions to lower levels, i.e., introduction to same or higher levels should also be possible, with the restriction that these introductions can not be covered by pointcuts of the introducing aspect.

4.6 Enabled Language Extensions

With our type-level language extension defined as above, we are now ready to extend AspectJ safely with constructs allowing the expression of aspects that advise, or do not advise, themselves.[11] For instance, a special variable `targetaspect` can now be introduced that refers to the (instance of) the aspect whose join point (as captured by an `adviceexecution()` pointcut) is currently handled. Another special variable `thisaspect` can be added that refers to the (instance of) the current (handling) aspect. Note that the type (level) of `thisaspect` is always the same as that of the advice in whose context (lexical scope) the variable occurs, while that of `targetaspect` is necessarily of a lower level; therefore, the expression of the aspect from Section 2.2 that advises all aspects that do not advise themselves,

```
aspect Barber {
  void around(): adviceexecution() {
    if (targetaspect != thisaspect) {
      proceed();
    else {}
  }
}
```

causes a type level mismatch error in line 3.

[11] A similar request for language extension has been formulated in [14].

5. DISUSSION AND RELATED WORK

5.1 Dependence of the Antinomies on a Declarative Interpretation

When reconstructing the logical antinomies in AspectJ in Section 2.1, we relied on what we called "intuitive semantics". This assumed intuitive semantics is basically a declarative one, i.e., we read the programs as assertions rather than as sequences of instructions. When looking at it with procedural glasses on, the advice of aspect `S` in Section 2.1 would read as "before executing any advice, call the advice of `S`". Since "any advice" includes itself, the advice of `S` is called recursively *before* anything else is (not) done. Therefore, one might argue that there is no antinomy in the program, just an infinite recursion. However, the reader will agree that this procedural semantics (as defined by the AspectJ compiler) is non-obvious at best, and that in a well designed language, intuitive semantics should match the operational one (the principle of least surprise).

As an aside, it is interesting to note that the procedural semantics of aspects allows them to avoid self-reference through the `cflow()` construct. In fact, in a purely declarative interpretation, particularly without a notion of sequentiality and without having access to the history of execution, an exclusion of self-reference cannot be formulated (as noted by Russell in the quote of Section 3.1). The price for this check is that it has to be done at runtime (and is in fact very expensive); by contrast, our type system allows a static check, which (in terms of runtime overhead) is entirely free.

5.2 Typing to Prevent Programming Errors

Even if one denies the existence of antinomies in the aspects constructed in Section 2.1, one will agree that a well-designed programming language should save its programmers from programming errors. In fact, type systems are generally accepted as serving this purpose; they discover many possible type mismatches at compilation time. However, the aspects of AspectJ, although syntactically similar to classes, are untyped; therefore, current typing systems cannot prevent any errors related to advice application. We have adapted a typing system well-suited for this purpose from Russell's theory of types and Tarski's theory of object language and meta-language; although it looks very different from that base language's (i.e., Java's) type system, it serves the same purpose: it prohibits the construction of illegal programs, and it prevents programming errors.

5.3 The Orderedness of AOPLs

It has been noted elsewhere that AOPLs are necessarily second-order languages [15]. Second-orderedness by definition excludes self-referentiality, so that in all AOPLs that are true second order languages (as are all those languages that exclude aspects from being applied to aspects), the above antinomies cannot be expressed. However, as we have demonstrated here, at least AspectJ as its stands is an unordered language; like unordered logic, it allows the construction of Russell's "vicious circles". From all we can see, making AspectJ a well-ordered ("typed") language as proposed here fixes the problems without imposing any undue restrictions.

5.4 Aspects of Aspects and the Closure of Languages

The notion of aspects of aspects has stirred some theoretical contemplation concerning the closedness of aspect languages. In [7], the authors state that the goal of any aspect language should be that it be "closed with respect to aspectification (aspect closure)". This is expressed by the equation $A(L) = L$, meaning that expressing aspect application to the language elements of L would make do with L, that is, would not require additional language elements. From this, they deduce that AspectJ as an instance of A(Java) is currently not closed, since obviously AspectJ ≠ Java, but also (currently) A(AspectJ) ≠ AspectJ. They argue in favour of such a closure since they observed that certain languages incorporating meta-programming, such as Smalltalk or CLOS, are also self-contained, i.e., that there $M(L) = L$. However, they ignore that this is only possible because these languages resort to certain tricks: for instance, in Smalltalk the class `MetaClass` is an instance of itself. This however forbids the semantic interpretation of classes as sets and instances as elements of sets, since then the set of `MetaClass` would have to contain itself. At the same time, it is at odds with Tarski's fundamental observation that the meta-language must be richer than its object language, meaning that it cannot be "semantically closed".

Returning to the generic `meta^n` keyword discussion from Section 4.3, we note that our language, although handled by a single compiler, is not semantically closed, since every new meta-level, i.e., $A(L)$ where the highest type level in L is n, requires a new keyword `meta^n`. In a similar vein, the syntax of predicate logic can encompass various orders (i.e., first order predicate logic, second order predicate logic, and so forth).

5.5 Testing Advice

In Ref. [14] it is argued that testing is a crosscutting concern, i.e., that testing code spreads across the whole system, and thus lends itself to being extracted to an aspect. All testing code can be encapsulated into one module which has privileged access to the original source without needing to modify it. Furthermore, the testing code is easily removed, by excluding it from compilation. As AOP seems to be well suited for testing object-oriented programs, the question arises whether it is also well suited for testing aspect-oriented programs.

To focus the discussion, the authors distinguish between the "application aspects", i.e. aspects applying to the base program for some application-specific purpose, and "testing aspects", i.e. aspect applying to the base program or application aspects for the purpose of testing them. With the aid of our type levels, one can syntactically separate these levels, by modifying testing aspects with `meta`. Also, our type-safeness opens the door for the extension of reflection mechanisms requested in [14] "so that information about actual join points, pointcuts and advices can be obtained", without worrying about new problems (*cf.* Section 4.6).

6. CONCLUSION

As a form of meta-programming, aspects and AOP are so powerful concepts that their use must be regulated. In particular, possible self-application of aspects is a severe problem, since it cannot only cause infinite recursion, but also allows nonsensical expressions. While the former should merely be prevented, the latter must be forbidden by any sound language definition. Based on the groundbreaking work of Russell and Tarski, we have proposed a simple language extension that equips AspectJ with a (formerly unavailable) typing system suitable to eliminate both kinds of problems through a simple static type check. Even though we based our argumentation on one specific implementation of AOP, the problem and the solution presented in this paper should be applicable to AOP in general.

7. REFERENCES

[1] AspectJ homepage, http://www.aspectj.org, 2006

[2] P. Avgustinov et al: "Optimising AspectJ", *Technical Report*, 2004

[3] http://www.eclipse.org/aspectj/doc/released/progguide/

[4] R.E. Filman et al: "Aspect-Oriented Software Development", *Addison-Wesley*, 2004

[5] F. Forster: "Points-To Analyse und darauf basierende Interferenzanalyse für eine Kernsprache von AspectJ", *Master Thesis*, University of Passau, 2005

[6] J.D. Gradecki, N. Lesiecki: "Mastering AspectJ", *John Wiley & Sons*, 2003

[7] K.B. Graversen and K. Østerbye: "Aspects of Aspects – a framework for discussion", *European Interactive Workshop on Aspects in Software*, 2004

[8] D.R. Hofstadter: "Gödel, Escher, Bach: an Eternal Golden Braid", *Basic Books Inc.*, 1979.

[9] http://www.jcp.org/en/jsr/detail?id=269

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold: "An overview of AspectJ", *Proceedings of ECOOP '01*, 2001

[11] G. Kiczales et al. : "Aspect-oriented programming", *Proceedings of ECOOP '97*, 1997

[12] http://en.wikipedia.org/wiki/Metaprogramming_(programming)

[13] B. Russell: "Mathematical Logic as Based on the Theory of Types", *American Journal of Mathematics Vol. 30*, 222-262, 1908.

[14] D. Sokenou and S. Herrmann: "Aspects for Testing Aspects?", *Workshop on Testing Aspect Oriented-Programms AOSD 2005*, 2005

[15] F. Steimann: "Domain Models are Aspect Free", *MoDELS/UML 2005 (Springer 2005)*, 171–185, 2005

[16] http://java.sun.com/j2se/1.5.0/docs/guide/apt/GettingStarted.html

[17] A. Tarski: "The Semantic Conception of Truth and the Foundations of Semantics", *Philosophy and Phenomenological Research 4*, 341-375, 1944