Avoiding Infinite Recursion with Stratified Aspects

Eric Bodden
Sable Research Group
McGill University
3480 University Street
H3A 2A7 Montréal, PQ, CA
bodden@acm.org

Florian Forster
LG Programmiersysteme
Fernuniversität in Hagen
Universitätsstraße 1
D-58097 Hagen
florian.forster@feu.de

Friedrich Steimann
LG Programmiersysteme
Fernuniversität in Hagen
Universitätsstraße 1
D-58097 Hagen
steimann@acm.org

Abstract: Infinite recursion is a known problem of aspect-oriented programming with AspectJ: if no special precautions are taken, aspects advising aspects can easily and unintentionally advise themselves. We present a compiler for an extension of the AspectJ programming language that avoids self reference by associating aspects with levels, and by automatically restricting the scope of pointcuts used by an aspect to join points of lower levels. We report on a case study using our language extension, and provide numbers of the changes necessary for migrating existing applications to it. Our results suggest that we can make programming with AspectJ simpler and safer, without restricting its expressive power unduly.

1 Introduction

AspectJ lets aspects advise aspects, including themselves. In previous work, we have argued on theoretical grounds that this self reference can lead to paradoxical expressions if an aspect changes its own meaning [FS06]. In programming practice, self reference of aspects, whether paradoxical or not, can lead to infinite recursion, and is almost always unintended, that is, a programming error [AT].

AspectJ programmers have identified unintended recursion as a common problem and have developed an idiomatic workaround: in order for an aspect `A` to not advise itself or methods that it calls, its pointcuts are conjoined with `!cflow(within(A))`. This ensures at runtime that the advice only executes if the current point of execution is not in the dynamic control flow of one of the join points in `A`. Although one can keep the necessary runtime overhead to a minimum by using smart compilation techniques [Av05b], it is in the responsibility of the programmer to add this check, which he may forget. In practice one often fails to discover the need for such guards until the debugging phase of a project.

In our previous paper, which was inspired by the works of Bertrand Russell and Alfred Tarski, we suggested the introduction of type levels to aspect-oriented programs [FS06]. More specifically, we suggested use of a `meta` modifier to tag aspects with the type (or meta) level they reside in, and to tag pointcuts with the level they apply to. By requiring that aspects can apply only to program elements from a lower level (a property that can be checked statically, i.e., at compilation time), we can avoid unintended recursion that derives from the self reference of aspects. A question that remains is whether this ap-

E Bodden, F Forster, F Steimann “Avoiding infinite recursion with stratified aspects” in: R Hirschfeld, A Polze, R Kowalczyk: *Proceedings of NODe* GI LNI P-88 (2006) 49–64.

proach also prohibits intended recursion. However, our only mildly successful search for practical examples of aspects that are to advise themselves suggests that such is a rare requirement. More frequent seem to be cases in which aspects are to advise each other mutually, but not themselves; this is indeed prevented by our approach, but, as shown in Section 4.3 and further argued in Section 6.4, can be made possible by a simple trick.

The contribution of this paper is twofold:

1. we present an implementation of our suggested system of "stratified aspects" for AspectJ in the AspectBench Compiler (abc) [Av05a], and
2. we demonstrate the feasibility of our approach by reporting on the refactoring of several existing AspectJ programs suffering from self-reference problems to adopt the new syntax.

As a side effect, we find that stratification of aspects is not only a useful concept for avoiding recursion, but is sometimes also a natural property of the crosscutting concerns they represent.

Our results indicate that the amount of refactoring necessary to adopt stratification is reasonable, ranging from simply removing `!cflow/within` expressions to additionally inserting a few `meta` modifiers and additional disjuncts in pointcut specifications. In addition, for one program, namely the Law-of-Demeter checker described in [LLW03], we show that a simple copy&paste transformation of aspects allows the checker to check itself. This is in contrast to the fact that the original developers of the checker experienced serious problems with avoiding recursion in making the checker check other aspects, and gave up on making it check itself (personal communication with Karl Lieberherr).

The remainder of this paper is organized as follows. First we use a small example to show how our language extension, named AspectJ* in the sequel, changes the way of dealing with infinite recursion caused by aspects. Then, we describe the AspectJ* language in general and its implementation in the abc compiler. In Section 4, we walk through a case study based on the aforementioned Law-of-Demeter checker. In Section 5, we quantify the cost and benefit of our approach, by reporting on the number of added and saved expressions in a small number of sample applications. In Section 6, we justify our language design, name its limitations, discuss our work in the context of others', and point to future work.

2 Introducing AspectJ*

To present our language extension and the problems it addresses, we resort to a simple example. Consider the following tracing aspect:

```
aspect Tracing {
  Object around(): execution(* *(..)) {
    System.out.println("Entering:" + thisJoinPoint);
    return proceed();
    System.out.println("Leaving: " + thisJoinPoint);
  }
}
```

Now we decide to refactor the aspect, by extracting `println(.)` to a separate method:

```
aspect Tracing {
  private void show(String msg) {
    System.out.println(msg);
  }
  Object around(): execution(* *(..)) {
    show("Entering:" + thisJoinPoint);
    return proceed();
    show("Leaving: " + thisJoinPoint);
  }
}
```

The problem with the so refactored aspect is that its pointcut matches the execution of `show(.)`, which it calls itself so that an infinite recursion results. (Note that were the aspect weaver allowed to weave into system libraries, the pointcut would also match `println(.)`. For the sake of this example, we assume common practice, i.e., that it is not.) To avoid the recursion, we can change the aspect's pointcut to

```
execution (* *(..)) && !within(Tracing)
```

In AspectJ* the additional condition in the pointcut is not necessary, because pointcut primitives such as `execution(.)` by definition only match join points on a lower level, in this case the class level, thereby avoiding all recursion arising from join points within aspects.

Next consider that we wish to extend our tracing aspect so that it also traces advice execution. For this, the pointcut of the aspect is extended to

```
execution(* *(..)) || adviceexecution()
```

However, this pointcut matches the execution of the advice using it, entailing another infinite recursion. Again, adding `!within(Tracing)` would help (assuming that `Tracing` was not meant to trace its own execution, not even once; we shall return to this in Section 4.3). However, in AspectJ* an aspect cannot advise aspects on the same level, including itself – in fact, `adviceexecution()` on the (unmodified) aspect level is a semantic error detected by the compiler. To fix it, i.e. to let `Tracing` advise other aspects, it must be raised to the meta level, which is done by modifying the aspect with `meta`, as in

```
meta aspect Tracing {
  …
}
```

Now we have one disjunct of the pointcut matching method executions on the class level (`execution(* *(..))`) and one disjunct matching advice execution (`adviceexecution()`). What is lacking is a disjunct matching method executions on the aspect level (i.e., methods that are defined within an aspect, such as `show(.)` in the non-meta aspect `Tracing` above). If desired, this behaviour can be added by writing

```
execution(* *(..)) || adviceexecution() || meta execution(* *(..))
```

instead of the above pointcut. Note that we do not modify `adviceexecution()` with `meta` because it applies to advice and thus is already one level above non-modified pointcuts (see Figure 1). Also note that AspectJ* allows us to express the targets of advice on a finer level than possible in AspectJ: for instance, by omitting the `meta execution (* *(..))` disjunct from the pointcut we can exclude the advising of methods defined within aspects; by omitting `execution(* *(..))` we can constrain the aspect's

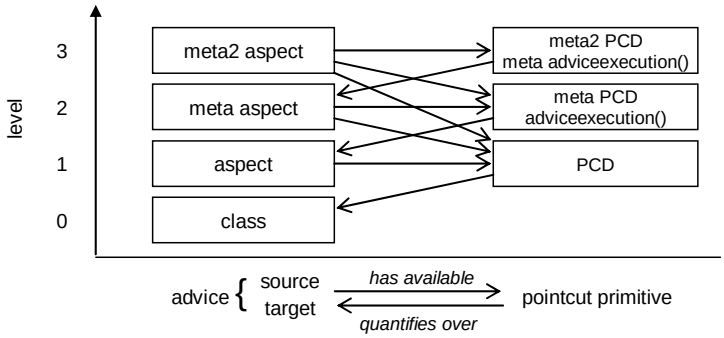

Figure 1: Stratified aspects, their admissible pointcut primitives, and their potential matches.

advising to advice and methods defined within aspects only (used in our case study in Section 4.3). In AspectJ, `execution(* *(..))` would match on classes and aspects and a finer granularity could only be achieved through naming conventions.

3 Implementation of AspectJ*

In order to evaluate the usefulness of our approach, we implemented the proposed syntax as an extension to the AspectBench Compiler (abc) [Av05a], an open compiler for language research in aspect-oriented software development. The extension consists of the following parts, which we describe in detail below:

- an extension to the lexer and grammar in order to recognize the new syntax;
- an extension to the type checker, ensuring that only valid uses of `meta` pointcuts pass the compilation; and
- an extension to the matcher, implementing the semantics of the `meta` modifier.

3.1 Syntax of AspectJ*

In order to accommodate our extension, we altered the existing AspectJ grammar in abc in the following way. First of all, we provided a mechanism to annotate aspects with meta levels. abc allows for the addition of new production rules via the `extend` keyword. The additional syntax is highlighted.

```
extend aspect_declaration ::=
    [modifiers] meta [level] aspect identifier
        [super_spec] [interface_spec] [perclause] aspect_body
level ::= "(" integer_literal ")" | ϵ
```

Consequently, each aspect definition can be preceded by a modifier `meta` (denoting meta level 1) or `meta(i)` for an integer `i`.[1] We used the same syntax to extend the definition of

[1] Note that meta level 1 corresponds to level 2 in Figure 1.

basic pointcuts. Basic pointcuts are all those pointcuts which have no inner pointcuts. This rules out `cflow` pointcuts, named pointcuts and pointcuts used for Boolean combinations.

```
extend basic_pointcut_expr ::= meta [level] basic_pointcut_expr
```

In addition, we had to modify the lexer so that it allows us to use integer constants in all positions in which the `meta` keyword may occur.

3.2 Type checking stratified aspects

In AspectJ* each well-typed aspect has to adhere to the following rules:

1. `meta(i)` is only legal for $i \geq 0$. In the following, `meta` is shorthand for `meta(1)` and absence of a meta modifier is equivalent to `meta(0)`.
2. `meta(i)` aspects may only extend `meta(i)` aspects. In particular, no aspect may extend a class. In programs in which this happens, the extended class has to be refactored to an abstract aspect.
3. `meta(i)` pointcuts may only be used in `meta(j)` aspects if $i \leq j$.
4. `meta(i) adviceexecution()` pointcuts may only be used in `meta(j)` aspects if $i < j$.

The first two rules are easy to check by standard mechanisms: we extended the type nodes for aspect declarations and basic pointcuts so that they can hold a meta level. The type checker was then altered to check for each aspect that its meta level is in the valid range and that it does not extend a class and, if it does extend an aspect, that this aspect has the same meta level.

The last two rules however were harder to handle correctly. The problem here is that it does not matter in which aspect a pointcut is *declared*: since pointcuts may be imported from other aspects or even classes, it is legal to declare them anywhere. However, they may only be *used* by advice on the correct level! For instance, the aspect `PointcutDatabase` in

```
aspect PointcutDatabase() {
  pointcut aspectMethodExecutions(): meta execution(* *(..));
}

aspect ShouldBeMeta{
   after(): PointcutDatabase.aspectMethodExecutions() {
      //the preceding line will raise a compilation error
   }
}
```

is allowed to declare `meta` pointcuts even though it is not itself on the `meta` level; however, `ShouldBeMeta` must be declared as a `meta` level aspect in order to use them.

This means that we cannot check type-correctness until advice matching takes place, because it is here where named pointcuts are inlined. Hence we had to insert our additional type check at an unusual location, namely in the `inline` method of the aspect info for pointcut nodes. (An “aspect info” is the semantic equivalent of each valid syntactic

AspectJ construct in abc.) At this place, we can conveniently refer to the aspect *using* the meta pointcut and hence issue meaningful error messages.

3.3 Extending the matcher

Extending the matcher proved relatively easy. First we had to assign a default meta level (aspect level for `adviceexecution()` and class level for any other basic pointcut) to any basic pointcut that was not given an explicit meta level by the programmer. We could easily achieve this with a visitor pass over the abstract syntax tree.

Then in the matcher, before checking whether or not a basic pointcut matches a given join point shadow, we first check if the meta level of this pointcut equals the one of the class/aspect we match against. Only if this is the case, we proceed by matching the actual pointcut.

We made our implementation available at http://www.sable.mcgill.ca/~ebodde/meta, including source, several test cases to try it out and the code of the case studies which follow. It should be noted that our extension to abc can easily be used by other extenders to build their tools on top of our extension, directly making use of the semantics of stratified aspects. We give an example for this in Section 5.3.

4 The Law-of-Demeter checker case study

In reaction to our theoretical work described in [FS06] the Law-of-Demeter checker described in [LLW03] was brought to our attention. It checks for a running program whether the Law-of-Demeter is adhered to on the object level, i.e., whether or not object references are used in such a way that the shortest available path for accessing another object (counted as the number of consecutive indirections) is no greater than one. The project consists of a small number of abstract classes and aspects that implement the checker, and of a base program for which the law is checked. As its authors reported (personal communication), all attempts to make the checker check itself resulted in infinite recursion. Therefore, the implementation of the checker uses the `!cflow/within` idiom to constrain its pointcuts to the base program, thus excluding the checker itself (consisting of a few aspects and abstract classes containing pointcut definitions, advice and helper methods) from being checked. This is done by grouping all checker classes and aspects in a package called `lawOfDemeter`:

```
package lawOfDemeter;

public abstract class Any {
  public pointcut scope():
    !within(lawOfDemeter..*)
    && !cflow(withincode(* lawOfDemeter..*(..)));
  …
  public pointcut MethodCallSite():
    scope()
    && call(* *(..));
  public pointcut MethodCall(Object thiz, Object targt):
    MethodCallSite()
    && this(thiz)
    && target(targt);
```

```
  // and so forth; set of pointcuts matching Java constructs
  // for which the Law of Demeter can be checked
}
…
public aspect Check {
  private pointcut IgnoreCalls():
    call(* java..*.*(..));
  …
  after(Object thiz,Object targt):
    Any.MethodCall(thiz,targt) && !IgnoreCalls() {
      if (!ignoredTargets.containsKey(targt) &&
        !Pertarget.aspectOf(thiz).contains(targt))
      { objectViolations.put(thisJoinPointStaticPart,
                    thisJoinPointStaticPart);
      }
    }
}
```

To avoid that the call to `objectViolations.put(.,.)` in the advice of `Check` is matched by the pointcut `Any.MethodCall(thiz, targt)` on which it depends, this pointcut explicitly excludes all join points from the `lawOfDemeter` package, by conjoining the pointcut `Any.scope()`.

4.1 Avoiding recursion without `!cflow/within`

The use of the `!cflow/within` idiom is necessary even in absence of `adviceexecution()` because the aspects contain join points, and make use of methods containing join points, that would otherwise be matched by the aspects' own pointcuts. This is so because the Law-of-Demeter checker implements a dynamic program analysis targeted at the use of general language constructs (rather than types of join points specific to the base application) and the control logic in the aspects is implemented using the very same language constructs. As a result, the checker would naturally attempt to check itself, leading to infinite recursion.

By using AspectJ* instead of AspectJ, the aspects of the checker and the abstract classes they extend (which have to be declared as abstract aspects in AspectJ*) are automatically one level above the base program, and the pointcuts they contain are automatically restricted to classes as the base. Therefore, the aspects cannot possibly apply to themselves, so that all occurrences of the `!cflow/within` idiom can be removed. Indeed, after dropping all references to `!cflow/within` (basically the pointcut `scope()`), the program ran as before, reporting the same number of Law-of-Demeter violations in the base program. Note that dropping the contained `!cflow/within` clauses is possible, but not necessary: because they all refer to the `lawOfDemeter` package and thus the aspect level, they become superfluous. On the other hand, using AspectJ* means that the unmodified checker cannot check other aspects, simply because they are on the same level as the checker. To remedy this, we have to move the checker aspects up one level.

4.2 Checking the Law-of-Demeter in Aspects and Classes

The modifications necessary to do this were minor. First, we prefixed all checker aspects with the `meta` modifier. Next, we added `adviceexecution()` primitives to the pointcuts

(to match the advice of aspects) and prefixed copies of the other used primitives (`execution(.)`, `set(.)`, etc.) with `meta` (to match corresponding join points in aspects). Note that had we preferred a simpler language design (as discussed in Section 6.1), we could have spared ourselves the change of pointcuts, albeit only at the price of a loss of flexibility, which will be needed in the next subsection.

The changed program, which checks the Law of Demeter for all classes and non-meta aspects, reads as follows:

```
public abstract class Any {
  public pointcut scope(): !within(lawOfDemeter..*)
     && !cflow(withincode(* lawOfDemeter..*(..)));
  …
  public pointcut MethodCallSite():
     call(* *(..)) || meta call(* *(..));
  public pointcut MethodCall(Object thiz, Object targt):
    MethodCallSite()
    && ((this(thiz)  && target(targt))
       || (meta this(thiz)  && meta target(targt)));
  …
}
…
public meta aspect Check {
  private pointcut IgnoreCalls():
    call(* java..*.*(..)) || meta call(* java..*.*(..));
  …
}
```

Note that, as a process issue, if one adds the `meta` modifiers to the pointcuts first, the type checking pass of the compiler issues an error message if any `meta` modifiers for aspects are forgotten.

4.3 Making the checker check itself

After having moved all aspects of the project one level up, there were no aspects left on the non-meta level that the checker could check. Therefore, we added the original checker aspects to the changed project. After the necessary renaming of packages (the `meta` modifier does not affect name or namespace of the modified aspects, so that an aspect and its copy modified with `meta` cannot coexist in the same namespace), the program compiled smoothly. However, running it did not only discover four additional violations of the Law of Demeter (all occurring in the aspects that were designed to check it), but also resulted in all violations in the base program being reported twice, because both the original checker and the meta checker checked it. We were however easily able to fix this, simply by removing all components quantifying over the class level in the pointcut definitions used by the meta checker. We were thus left with

a) a checker that checks the Law of Demeter for base programs (excluding aspects) and

b) an almost identical checker that checks the Law of Demeter for all non-meta aspects.

Quite obviously, unless we explicitly want the checker to check itself (or, rather, a copy of itself), we would prefer the variant described in the previous subsection, i.e., to have one checker that checks base programs and aspects (but not itself). For the creation of aspects that can advise themselves (without running into an infinite recursion), we propose a more systematic approach in Section 6.4 as future work.

Finally, note that the same principle of letting the Law-of-Demeter checker check a copy of itself could have been realized in plain AspectJ; however, this would have required a number of additional `!cflow/within` expressions.

5 Refactoring to stratified aspects

The previous sections suggested that refactoring programs so that they make use of stratified aspects is not a big deal. Here, we briefly recapitulate what is necessary to turn AspectJ programs into AspectJ* programs of same functionality.

5.1 The refactoring procedure

As indicated above, refactoring existing programs to rely on stratified aspects (that is, to turn AspectJ programs to AspectJ* programs) is relatively easy:

1. For all programs in which no aspect advises other aspects, nothing needs to be done. Remaining (i.e., still undiscovered) sources of infinite recursion resulting from self-reference of aspects are automatically disabled, and for those that have been prevented by using the `!cflow/within` idiom, the guards can be removed.
2. For all programs in which aspects advise other aspects, the aspect-advising aspects must be moved up one level, by modifying them with the `meta` modifier. If these aspects advise only aspects and no classes, each of the pointcuts they refer to (except those involving `adviceexecution()`) must also be modified with `meta`. If these aspects advise aspects *and* classes, each pointcut disjunct must be duplicated, and the duplicate must be modified with `meta`.
3. For all programs in which two or more aspects should advise all others, but not themselves, as well as for programs in which one or more aspects should advise all aspects (including themselves), we have to refer the reader to our future work (described in Section 6.4).

We expect the majority of cases to be of category 1. First evidence justifying this expectation is presented next.

5.2 Cost and benefit of the refactoring

Now that we have seen that refactoring to AspectJ* requires a variable number of necessary steps, it would be interesting to know how much work we have to expect in the average case, and what the expected saving is. Therefore, we have conducted further case studies.

In order to identify projects in which aspects deliberately advise aspects, we looked for programs containing the `adviceexecution()` primitive. Unfortunately, we only found three. (The low availability of non-trivial AspectJ programs for benchmarking and other experiments is a known problem of the still young community.) Therefore, we added to our survey a few other programs (not using the `adviceexecution()` primitive) of which

we knew that they suffered from recursion induced by self-application of aspects. The results of refactoring these projects to AspectJ* are summarized in the following table and discussed in detail below. (See Appendix for references to the sources of the projects.)

	project:	tracing from AspectJ distribution	Glassbox Inspector	tracing from Glassbox Inspector	Design-by-Contract checker	Tetris	banking application	Law-of-Demeter checker[§]
	number of:							
project size	classes (total)	$n^{\dagger}$	15	n	n	8	2	n+1/+1/+2
	aspects[*]	2	24	1	1	8	3	4/4/8
before refactoring	occur. of `adviceexecution()`	0	2	1	$m^{£}$	0	2	0/1/1
	scope limitations using `within`[‡]	3	15	1	m	4	2	9/10/9
after refactoring	scope limitations using `within`	0	12	2	0	2	0	0/0/0
	meta modifiers	0	20/3[$]	1	0	0	0	0/18/18
	additional pointcut disjuncts	0	18/0	1	0	0	0	0/16/2

§ for the variants checking classes only, checking classes and aspects excluding itself, and checking everything (by means of two stacked aspects, each extending only to the level immediately below it)

† number of classes depends on the particular base application, which can be swapped

* including abstract aspects and classes extended by aspects

£ one per assertion to be checked

‡ including `!within`, `!cflow/within`, etc.

$ smaller numbers obtained by letting unmodified pointcuts cover all lower levels (see text)

Most notably, almost all explicit scope limitations (using some variant of `within`) could be avoided. Those that remained represent explicit restrictions of aspects to classes or packages (Glassbox Inspector), or resulted from the declaration of warnings for certain procedure calls from within specific packages (Tetris). This suggests that using AspectJ* one can successfully mitigate the problem of structural pointcut matching based on naming conventions [SG05]; AspectJ* is one step into the direction of using more semantic matching constructs.

Another interesting observation is that meta aspects and additional pointcut disjuncts were only required in the refactored version where `adviceexecution()` occurred in the original code. Interestingly, although both the Design-by-Contract checker and the banking application did contain `adviceexecution()`, this was only used to *prevent* the containing pointcuts from matching advice and could be dropped in the corresponding AspectJ* program, requiring no additional modifications. Note that the relatively high first numbers of necessary pointcut modifiers (20) and additional disjuncts (18) for the Glassbox Inspector resulted from our language design decision to require pointcuts to be ex-

plicit about which level they apply to. Had we adopted a policy of letting all pointcuts used by an aspect on level *n* automatically apply to join points of all levels from 0 to *n*–1 (as discussed in Section 6.1), only the second number of changes (3 and 0) would have been necessary. Last but not least, the relatively high numbers of required meta modifiers and additional pointcut disjuncts given for the second and third version of the Law-of-Demeter checker reflect its altered behaviour: the second version extends checking to other aspects, and the third extends it to everything (including itself).

As an aside, when refactoring the Glassbox Inspector we noted a natural stratification of aspects. The Glassbox Inspector distinguishes between general aspects, aspects for logging, and aspects for error handling. All general aspects are advised by the logging aspects, which explicitly exclude themselves and the error handling aspects from being logged. All general and the logging aspects in turn are advised by the error handling aspects, which do not advise themselves. By using AspectJ*, this implicit hierarchy of aspects is naturally reconstructed in explicit form: error handling is a meta meta (`meta(2)` in AspectJ* syntax) aspect, logging is a meta aspect, and all others are ordinary aspects. Note that a similar observation has been made in [RS05]; it is discussed in Section 6.3.

One might argue that the moderate effort necessary to perform the refactoring is relativized by the moderate gains that can be expected. We counter this argument by stressing that our focus is not on refactoring. With using AspectJ*, infinite recursion resulting from self reference need not be avoided – it can no longer occur. Therefore, we suggest that programmers consider stratifying their aspects right from the beginning of their projects.

5.3 Savings for generated aspects

AspectJ is not only used as a source programming language, it can also be the target language of code generators. Such generators instrument ordinary (including non-ao) programs, e.g. for the purpose of dynamic analysis or runtime verification. Programs developed in this field often consist of two parts: a structured instrumentation engine and a backend which processes an event stream generated by the instrumentation. At least two of those programs, namely the runtime verification tools J-LO [Bo05, SB06] and EAGLE [Ba04] conduct their instrumentation by reducing an original (temporal) requirements specification to AspectJ aspects. For those aspects it is a natural requirement that they should not instrument themselves, but only the base application.

We wanted to evaluate if stratified aspects could be used in order to prevent unintended recursion automatically. Indeed, when J-LO was first developed by us about one year ago, we did experience such recursion and so had to alter our code generation strategy in order to include `!within` conjuncts. Adding such constructs during code generation is particularly painful because one operates on an abstract syntax tree (AST) rather than on a textual representation of the AspectJ program. Methods adding AST nodes for such purposes can easily occupy a full page of code. Moreover this is a redundant task for all such verification tools and it would be very beneficial, could it be avoided.

Fortunately, J-LO itself is also implemented as an extension to abc. By making J-LO extend AspectJ* instead of AspectJ within abc, we got the semantics of AspectJ* for free for the aspects generated by J-LO, preventing unintended recursion automatically. As a J-LO developer, we simply *knew* that any aspect generated by J-LO was safe with respect to that matter and this without writing any line of code in order to achieve this property, since those semantics are entirely handled by the AspectJ* extension. We conclude that that stratified aspects and AspectJ* are a good basis for the design of aspect code generators.

6 Discussion

6.1 Language design decisions

In the design of our language extension, we had to take several choices. Firstly, we had to decide whether or not the pointcuts used by an aspect should be explicitly meta-modified, or whether they should derive their type level from the aspect using it (which would be closer to our role model, Russell's type theory). Although the latter would have simplified the language considerably, it would have meant that pointcuts must *always* apply to *all* levels below that of the aspect, which can sometimes be contrary to what one wants. For example, it would be impossible to express that an aspect advises only other aspects (including the methods defined within these aspects), and no classes (which is also impossible to specify generically, i.e., without resorting to naming conventions, in current AspectJ).

Another design decision is that we do not allow subtyping across levels. In particular, this means that an aspect cannot inherit from a class. While we do not consider this to be a big loss (what does it mean for an aspect to specialize a class? is the class a proper abstraction of an aspect?), it means that AspectJ* is not backward compatible to AspectJ. However, letting inheritance cross levels would have bloated our typing system with numerous exceptions to its simple rules.

Last but not least, we decided to make no attempts to avoid recursion resulting from an aspect accessing members of a lower level. This will be discussed next.

6.2 Where we must fail: limitations of our approach

We cannot avoid all recursion with our approach. In particular, we cannot avoid recursion resulting from an aspect accessing elements of lower levels (including classes), because for us, these accesses are indistinguishable from accesses by program elements on the same level, which are to be advised. Such can only be achieved by distinguishing types of join points on the same (meta) level as described in [FS] (see also related and future work below), or by explicit guards such as `!cflow/within` expressions, which we wanted to avoid. However, our compiler could produce a warning wherever an aspect crosses levels by accessing elements from a lower level and this may result in recursion. In fact, the cflow analysis [Av05b] built into abc could be used to statically compute those warnings.

Another problem of our approach arises when different concerns are to be implemented that are to advise each other. For instance, it could be that we want the Law-of-Demeter checker and an aspect debugger implemented as an aspect to coexist, mutually advising each other, but not themselves. Incidentally, this was not the case for all applications that we looked at, presumably because they were all proof-of-concept of an implementation for a single concern. In practical scenarios however, this may well be the case. We seek to further explore this problem in future work (Section 6.4).

6.3 Related work

Open Modules [Al05] provide means to decompose an aspect-oriented piece of software into multiple modules in such a way that certain assumptions can be made about their interaction. In particular, for a join point to be visible to a given aspect, this join point has to be explicitly exposed to it by a module interface specification. The idea differs from ours in that Open Modules provide a means for decomposing the set of join points of an application horizontally, without resorting to (vertical) layers. Therefore, the two approaches appear to be orthogonal to each other. The question is whether the modules of Open Modules can also be used to prevent unintended recursion. Because internal join points are matched without an explicit export required, we do not see how this could be achieved for direct recursion. Indirect recursion on the other hand, involving two or more aspects that are not "friends of" [On06] the same module, can be prevented, simply by avoiding circular dependencies between modules. However, we doubt that the desire to avoid unintended recursion would really justify the separation of a system into modules.

Open Modules were only recently integrated in the abc compiler [On06]. An interesting question is now whether Open Modules can emulate stratified aspects. For this, it would be necessary that a module can export its join points selectively to certain other modules. The classes and aspects of an application could then be decomposed into modules rather than levels, and each module could make sure that exactly the modules representing higher levels have access to its contained pointcuts, simply by exporting to those only. The resulting interface specification would be massive (because all join points must be exported) and extremely sensitive to changes within the module. Also, interface specifications could not be reused (or shared) among different levels, since an aspect on a higher lever must be able to distinguish between join points from different lower levels. All in all, even if the simulation of levels might be possible using Open Modules or some minor variation thereof, such is clearly not in the spirit of the approach. In our opinion, a good symbiosis would be to have base code and non-meta aspects organized by open modules and then address integration concerns (as described in [RS05]) by using meta-aspects which organize those modules.

In [St05] the last author of this paper has argued that aspects are second-order constructs, a view that is not universally shared in the AOP community. However, it works in favour of our approach, which views aspects as concepts confined to the second and higher orders. There are of course other, more advanced type systems for programming languages that we could have looked into. However, we believe that the one we have sug-

gested suffices for our purposes, and because it is also rather simple (and should thus be quickly embraced by programmers, which is not necessarily the case for all type systems!), we see no reason to replace it for another.

In [RS05], Rajan and Sullivan also argued in favour of hierarchical layering of aspects, so as to be able to cleanly modularize what they call higher-order crosscutting concerns: "In effect, [in AspectJ] higher-order concerns are all squashed down into – and consequently scattered across and tangled into – the single available aspect layer." [RS05] In their proposed programming language Eos-U, they (re)unite the class and aspect concepts to so-called classpects, representing aspects as objects (rather than classes) that can be bound to others, acting as their advisors. Advice is replaced by ordinary (named) methods, which are bound to join points of the advised objects using the pointcut expression syntax from AspectJ. Because advisors can advise advisors, hierarchies of aspects (or, rather, aspect objects) can be created. However, these hierarchies are expressed dynamically, through the composition of objects; in particular, no hierarchy on the type level is induced. Hence no guarantees about possible recursion can be given at compile time.

Accidental recursion due to a conflation of meta-levels has also been observed by Chiba et al. [CKL96]. The authors maintain that if a programming language with a meta object protocol fails to provide means of separating meta-levels explicitly, meta-circularity of the programming language may lead to control-flow circularity (i.e., recursion) on the implementation level. To solve this problem, the authors introduce an *implemented-by* relationship that complements the usual *instance-of* and *subclass-of* relationships of object-oriented programming by one that captures "the relationship between implemented and implementing objects and class metaobjects" [CKL96]. Translated to AOP, this *implemented-by* relationship could be interpreted as an *advises* relationship between an aspect and its base. Interestingly, the authors of [CKL96] perceived the introduction of a "special purpose test that prevents the infinite recursion" as seeming ad hoc, possibly difficult to reason about, and not effective in general. This is in contrast to the fact that currently, AspectJ (on whose design at least one of the authors of [CKL96] has considerable influence) offers only such a test (namely `!cflow/within` guards) for avoiding circularity (or self-referentiality [FS06]) in aspect-oriented programs.

6.4 Future work

Although aspects that should advise themselves are rather rare, our copy&paste with meta approach, which effectively limits the levels of recursion to one, is only a workaround. One possibility to avoid it would be to parameterize aspects on the meta level, i.e., to use a type level variable `N` in the definition of an aspect, as in

```
meta[N] aspect Tracing …
```

Like a template in C++, the variable `N` could be instantiated to yield an aspect at the specified level. The compiler would then produce a copy of this aspect with the type variable replaced by its value. However, this would require that the levels of the pointcuts used by the aspect are also parameterized. Since the levels are not necessarily identical to that of the aspect, relative levels would have to be specified, as in

```
meta [N-1] pointcut …
```

This however implies that instantiation of a template can fail, namely when a relative level drops below zero. While this could be avoided by dismissing explicit pointcut levels altogether (as discussed in Section 6.1), we want to collect more practical experience with the use and usefulness of meta aspects in general, before we address this particular problem.

A variant of the problem of the aspect that should advise itself, but only once (i.e., without running into a recursion), is that in which two or more aspects should mutually advise each other, but not themselves (an example is mentioned in Section 6.2). This is also problematic, because indirect recursion can be the result. The aspects cannot be brought into appropriate order, since whichever resides a level above the other(s) cannot be advised by the other(s). The solution is analogous to what we have proposed above: simply instantiate the aspects for two consecutive levels. In fact, this workaround may be needed so often that it might be useful to extend the language so that the compiler generates the meta-aspects automatically, for instance directed by using `self` as an (additional) modifier. However, more experiments have to be done here.

For an alternative solution, we could once again resort to mathematical logic. The type levels of Russell, although fully sufficient to avoid self reference, can be generalized to sorts, i.e., to partitions of the universe of discourse that are independent of levels. In particular, sorts allow partitions on the same level. If we can use sorts (or types) to explicitly declare sets of join points on the same level, and if we declare aspects with the types of join points which they advise, we can also allow cross application of aspects on the same level, avoiding recursion, albeit only at the price of obliviousness [FF04]. The implementation of a corresponding proposal [FS] is still pending, as is the unification of the corresponding type system with the one described here.

7 Conclusion

AspectJ programmers often face the problem that infinite recursion can result from unanticipated pointcut matches. Not infrequently, the recursion is only discovered at the debugging stage of a project, and then avoided by adding a programming idiom that tests the call stack. In [FS06], we suggested a more principled approach to avoiding infinite recursion, or letting it get caught by the compiler, by introducing levels of aspect application.

In this paper, we have presented an extension of the abc compiler that realizes our language extension, named AspectJ*, and discussed the cost and benefit of using stratified aspects to replace for explicit guards. For this, we have collected data from all available projects that use `adviceexecution()` as a potential source of infinite recursion, and a few others that suffer from self-reference through other pointcuts. By applying it to the Law-of-Demeter checker described in [LLW03], we demonstrated for a specific example how AspectJ* simplifies program design, and makes aspect applications – including self application – simple that were previously thought to be too difficult to get right.

References

[Al05] J Aldrich "Open Modules: modular reasoning about advice" in: *ECOOP 2005* Springer LNCS 3586 (2005) 144–168.

[AT] AspectJ Team *The AspectJ™ Programming Guide* Chapter 5: Pitfalls http://www.eclipse.org/aspectj/doc/released/progguide/index.html

[Av05a] P Avgustinov et al. "abc: an extensible AspectJ compiler" *Transactions on Aspect-Oriented Software Development* (2005) 293–334.

[Av05b] P Avgustinov et al. "Optimising AspectJ" in: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2005) 117–128.

[Ba04] H Barringer, A Goldberg, K Havelund, K Sen "Program Monitoring with LTL in EAGLE" in: *18th International Parallel and Distributed Processing Symposium, Parallel and Distributed Systems: Testing and Debugging - PADTAD'04* (IEEE Computer Society Press 2004).

[Bo05] E Bodden *J-LO – A Tool for Runtime-Checking Temporal Assertions* Diplomarbeit (RWTH Aachen 2005).

[CKL96] S Chiba, G Kiczales, J Lamping "Avoiding Confusion in Metacircularity: The Meta-Helix" in: *2nd International Symposium Object Technologies for Advanced Software* Springer LNCS 1049 (1996) 157–172.

[FF04] RE Filman, DP Friedman "Aspect-oriented programming is quantification and obliviousness" in: RE Filman et al. (eds) *Aspect-Oriented Software Development* (Addison-Wesley 2004).

[FS] F Forster, F Steimann "Programming with join point types and polymorphic pointcuts" unpublished manuscript *http://www.fernuni-hagen.de/ps/pubs/PolymorphicPointcuts.pdf*.

[FS06] F Forster, F Steimann: "AOP and the antinomy of the liar" in: *Workshop on the Foundations of Aspect-Oriented Languages (FOAL)* @ AOSD (2006).

[LLW03] KJ Lieberherr, DH Lorenz, P Wu "A case for statically executable advice: checking the law of demeter with AspectJ" in: *AOSD* (2003) 40–49.

[On06] N Ongkingco et al. "Adding Open Modules to AspectJ" in: *AOSD* (2006) 39–50.

[RS05] H Rajan, KJ Sullivan "Classpects: unifying aspect- and object-oriented language design" in: *ICSE* (2005) 59–68.

[SB06] V Stolz, E Bodden "Temporal assertions using AspectJ" in: *Fifth Workshop on Runtime Verification (RV'05)* ENTCS 144:4 (2006) 109–124.

[SG05] M Störzer, J Graf "Using pointcut delta analysis to support evolution of aspect-oriented software" *21st IEEE International Conference on Software Maintenance* (2005) 653–656.

[St05] F Steimann "Domain models are aspect free" in: L Briand, C Williams (Eds) *MoDELS/UML 2005* Springer LNCS 3713 (2005) 171–185.

Appendix: URLs of the refactored projects

Tracing from AspectJ distribution: http://www.eclipse.org/ajdt/

Glassbox Inspector: https://glassbox-inspector.dev.java.net/

Tetris: http://www.sable.mcgill.ca/benchmarks/

Banking application: http://www.cs.hofstra.edu/~cscccl/csc123/aop/aopbank.java

Design by Contract checker: http://www.cs.tau.ac.il/~ohadbr/aop/DbcUsingAOP.ppt

Law-of-Demeter checker: http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/AspectJCheckers/PaperObjectForm/