The abc Group

Building the abc AspectJ compiler with Polyglot and Soot

abc Technical Report No. abc-2004-2

Pavel Avgustinov[1], Aske Simon Christensen[2], Laurie Hendren[3], Sascha Kuzins[1], Jennifer Lhoták[3], Ondřej Lhoták[3], Oege de Moor[1], Damien Sereni[1], Ganesh Sittampalam[1], Julian Tibble[1]

[1] Programming Tools Group, Oxford University, United Kingdom

[2] BRICS, University of Aarhus, Denmark

[3] Sable Research Group, McGill University, Montreal, Canada

October 21, 2004

Contents

List of Figures

List of Tables

Abstract

Aspect-oriented programming and the development of aspect-oriented languages is rapidly gaining momentum, and the advent of this new kind of programming language provides interesting challenges for compiler developers. Aspect-oriented language features require new compilation approaches, both in the frontend semantic analysis and in the backend code generation. This paper is about the design and implementation of the *abc* compiler for the aspect-oriented language AspectJ.

One important contribution of this paper is to show how we can leverage existing compiler technology by combining Polyglot (an extensible compiler framework for Java) in the frontend and Soot (a framework for analysis and transformation of Java) in the backend. In particular, the extra semantic checks needed to compile AspectJ are implemented by extending and augmenting the compiler passes provided by Polyglot. All AspectJ-specific language constructs are then extracted from the program to leave a pure Java program that can be compiled using the existing Java code generation facility of Soot. Finally, all code generation and transformation is performed on Jimple, Soot's intermediate representation, which allows for a clean and convenient method of applying aspects to source code and class files alike.

A second important contribution of the paper is that we describe our implementation strategies for the new challenges that are specific to aspect-oriented language constructs. Although our compiler is targeted towards AspectJ, many of these ideas apply to aspect-oriented languages in general.

Our *abc* compiler implements the full AspectJ language as defined by *ajc* 1.2.0 and is freely available under the GNU LGPL.

1 Introduction

Aspect-oriented programming is rapidly gaining popularity and AspectJ [9] is widely recognised as one of the key aspect-oriented programming languages in use today. To date, there has been only one compiler for AspectJ — *ajc*, originally developed by the inventors of AspectJ at Xerox PARC [14] and currently developed and maintained as part of the Eclipse AspectJ project [2].

This paper is about the design and implementation of a new compiler for AspectJ, the *AspectBench Compiler*, *abc* [1]. Our original motivation for building *abc* was to create a workbench that allows easy experimentation with new language features [3] and new optimisations. However, we found that implementing an alternative compiler also helped to clarify the language semantics. Further, for users of a language, it is often useful to have different compilers, as each compiler has it strengths, and it provides a way for users to verify that they are not relying on any implementation-specific behaviour.

As researchers in the compiler field we felt that it was important for us to leverage previous work in the area of compiler toolkits for building Java frontends and backends. Thus, an important contribution of this paper is to show how we combined the Polyglot framework for extensible Java frontends [13] with the Soot framework for analysis and optimisation of Java [19]. Combining the tools was a non-trivial challenge, and a substantial part of *abc*'s architecture design stems from the need to cleanly separate the Java part of AspectJ programs from the aspect-specific parts in a way that can be used by both the frontend and backend Java tools. We are the first AspectJ compiler to achieve a clean separation of the implementation of the aspect-oriented features from these underlying tools.

Current versions of the AspectJ language specify that weaving (injecting aspect code) should be done at the bytecode level (as opposed to source level), and input to the compiler can be AspectJ/Java source, or Java class files. Designing *abc* to handle both kinds of inputs in a natural way was another important design challenge.

Implementing compilers for aspect-oriented languages is a relatively new field, and another important contribution of this paper is to show the structure of such a compiler, and to describe how we have implemented the various parts that are specfic to compiling aspect-oriented code. Although *abc* is an AspectJ compiler, many of these components would also be necessary for other aspect-oriented languages.

The structure of this paper is as follows. In Section 2 we provide a brief introduction to the most relevant features of AspectJ.[1] In Section 3 we briefly summarise our building blocks, Polyglot and Soot. Section 4 provides a description of the architecture of *abc*, and how this architecture fits together with our building blocks. Section 5 discusses details of how specific aspect-oriented features have been addressed, namely how we handle *name matching*, *declare parents*,

[1] We assume that many compiler researchers are not yet familiar with AspectJ, readers with previous knowledge of AspectJ may skip this section.

intertype declarations and *advice*. Finally, Section 6 reviews related work and Section 7 gives conclusions and future work.

2 An overview of AspectJ

An AspectJ program consists of two kinds of entities: ordinary Java classes and *aspects*, which are instructions for injecting code into the classes at specific points and under specific conditions. Aspects are applied to classes (and the aspects themselves) by a process known as *weaving*: an AspectJ compiler reads in the aspects and classes to be compiled and produces classes in which the aspect code has been injected as specified in the aspects.

To introduce AspectJ's features, we have chosen a small expression interpreter in Java, to which we will apply five illustrative aspects. As illustrated in Figure 1(a), most of the interpreter was generated using the SableCC parser generator, and the generated code is in four packages providing the lexer, parser, tree nodes, and various tree traversal visitors. In addition to the generated code there are two small programmer-defined Java classes: `tiny/Main.java` contains the main method which reads in input, applies the parser and then evaluates the resulting expression tree. The actual expression evaluation is performed by the method `eval` defined in the class `tiny/Evaluator.java`. An example of running the `tiny` interpreter is given in Figure 1(b).

Generated packages:
(must not be directly modified)

```
    lexer/
    parser/
    node/
    analysis/
```

User-defined package:

```
    tiny/Main.java
        /Evaluator.java
```

(a) code base

```
> java tiny.Main
Type in a tiny exp:
3 + 4 * 6 - 7
The result of evaluating:
3 + 4 * 6 - 7
is: 20
```

(b) example run

Figure 1: `tiny` interpreter example

The AspectJ language can be divided into *static* and *dynamic* features. Static features are defined and implemented with respect to the static structure of a program, whereas dynamic features relate to the dynamic trace of a program execution. Figure 2 shows the five example aspects which we apply to our example `tiny` interpreter code base.

2.1 Static Features

The `StyleChecker` aspect in Figure 2 illustrates an interesting use of AspectJ, the *declare warning* construct[2]. This construct allows the programmer to specify a pattern and a warning string. For each place in the program matching the pattern, a compile-time warning is issued, using the string as the warning message. In our example we have specified a pattern that matches all places where a field is set, and which are not within a method whose name starts with "*set*". In fact, the pattern is a bit more precise than this, because it will only match sets to non-private, non-final fields. When we compile the `tiny` code base with the `StyleChecker` aspect (`abc StyleChecker.java */*.java`) several warnings are given, mostly relating to the generated parser code, for example:

```
parser/TokenIndex.java:14: Warning --
                           Set of field outside of a set method.
        index = 0;
        ^-------^
```

When using SableCC (or other tools) to generate compilers, it is very important not to modify the generated code, so that it can be regenerated without clobbering the user's changes. SableCC generates all the classes representing the AST, with class `node.Node` as the root (extending `Object`), and a hierarchy of subclasses for other kinds of nodes

[2] There is also an analogous *declare error* construct

```
public aspect StyleChecker {
   declare warning :
      set(!final !private * *)
      && !withincode(void set*(..) ):
      "Set of field outside of a set method.";
}
```

```
public class Value {
   private int value; // a new field
   public void setValue(int v)
        { value = v; }
   public int getValue()
        { return value; }
}
public aspect ValueNodeParent {
   declare parents:
        node.Node extends Value;
}
```

```
public aspect AddValue {
   private int node.Node.value;
   public void node.Node.setValue(int v)
        { value = v; }
   public int node.Node.getValue()
        { return value; }
}
```

(a) static features

```
public aspect CountEvalAllocs {

   int allocs; // counter

   pointcut mainEval() :
      call(* *.eval(..)) && within(*.Main);

   before () : mainEval()
      { allocs = 0; }

   after () : mainEval()
      { System.out.println(
         "*** Eval allocations: " + allocs); }

   before () : cflow(mainEval()) && call(*.new(..))
      { allocs ++; }
}
```

```
public aspect ExtraParens {
   String around() :
     execution(String node.AMultFactor.toString()) ||
     execution(String node.ADivFactor.toString())
     { String normal = proceed();
        return "(" + normal + ")";
     }
}
```

(b) dynamic features

Figure 2: Illustrative AspectJ examples

below `node.Node`, as indicated by the grammar specification. This hierarchy is fixed in the generated code and since one should not edit these generated classes, it is not possible to add new fields to the nodes. The recommended method is to associate values with nodes using a hash table. However, using static features of AspectJ there are two ways of adding fields, without touching the generated code, without resorting to the use of external hash tables, and giving full semantic checking of the added fields.

The aspect `ValueNodeParent` from Figure 2(a) illustrates the AspectJ *declare parents* construct. In this example the programmer defines an ordinary class, `Value`, to implement the new field and accessor to that field. Then, the *declare parents* construct is used to inject the new `Value` class as a parent of the generated `node.Node` class. In general, the *declare parents* construct can be used to introduce new *extends* and *implements* relations.

Sometimes it is not possible (or desirable) to add new fields and methods by injecting new classes into the hierarchy, and AspectJ provides a general form of injecting new fields, constructors and methods into classes and interfaces, called *intertype declarations* or ITDs. The aspect `AddValue` in Figure 2(a) illustrates ITDs for injecting a new field and two new methods into the `node.Node` class. The declarations look like normal Java declarations, but the name of the field/constructor/method being defined is prefixed by the name of the class/interface into which it should be injected (in our example `node.Node`). Since AspectJ also allows one to inject new members into both classes and interfaces, ITDs can be quite powerful (and tricky to implement correctly in a compiler).

2.2 Dynamic Features

The dynamic features of AspectJ are quite different from the static features. While the static features are merely new incarnations of old ideas (in particular ITDs are a form of open classes), the dynamic features are generally regarded as the defining characteristic of aspect-orientation. They are defined with respect to a trace of the program execution. This trace is comprised of various kinds of observable events, for example: getting/setting fields, calling methods/constructors and executing method/constructor/intialiser bodies. These events may correspond to exactly one instruction (for example, getting/setting fields), or they may correspond to a group of instructions (for example, the body of a method). Each event has a starting point in the trace (just before it happens), and an ending point (just after it happens). The dynamic features of AspectJ allow one to specify a pattern to match certain events, and then advice (extra code) to execute *before*, *after* or *around* the matching events. The events are usually called *join points* in the literature on aspect-oriented programming, because these are places during program execution where an aspect can join in.

The aspect `CountEvalAllocs` in Figure 2(b) demonstrates *before* and *after* advice. The purpose of this example is to count the number of allocations that occur during the evaluation of an expression, starting from the call to `eval` in the `Main` class. In this example we define a pointcut `mainEval`, to specify that the *call* must be to a method called `eval`, and this call must occur *within* the `Main` class. Then we define *before* advice to initialise a counter just before the call, and *after* advice to print out the value of the counter just after the call. The tricky part of this aspect is the *before* advice used to increment the counter. We use the AspectJ *cflow* construct to specify that we are interested in all events that occur between the start and end of call to eval (i.e. between the time the call starts, and when the call is finished). We use the && operator to pick out, from those events, all events that call a constructor, as indicated by the pattern "**call**(*.**new**(..))". The *cflow* construct is of particular interest, because it means that we can match according to some runtime context, and because this matching cannot always be decided statically, runtime checks are necessary. There exist a number out other pointcut primitives (not covered in this introduction) that also require such runtime checks.

The `ExtraParens` aspect contains very simple example of *around* advice. This example is intended to slightly modify the output of the pretty print of expressions, by inserting parentheses around each factor. For example if the base program is compiled with this aspect (`abc ExtraParens.java */*.java`), the pretty print of the output in Figure 1(b) would be changed to `3 + (4 * 6) - 7`, instead of `3 + 4 * 6 - 7`.

The advice declaration in the `ExtraParens` aspect specifies a pattern to capture the execution of the two relevant methods. In the advice body, the *proceed* construct is used to specify that the original method body should be executed, the parentheses are added to the result, and this new result is then returned.

In our example the *proceed* call is very simple, but in general the use of *proceed* can be quite complex — it can be left out entirely, executed conditionally, called many times, saved for later execution using a local class, and the arguments can be modified. Thus, *around* advice is quite a bit more complicated than *before* and *after* advice, as it is not just injecting advice (code), but can actually change how existing code executes.

Also, it should be noted that all of our advice examples are very simple, and do not have any parameters. In general, advice may have parameters, and the pointcut patterns may specify how to bind those parameters to values. Readers who wish to know more details of the AspectJ language and its applications may wish to consult one of the growing number of textbooks on the subject, *e.g.* [11].

3 Building Blocks

In the following sections, we briefly introduce the building blocks of *abc*, Polyglot and Soot, focusing on the features that are most relevant to the *abc* design.

3.1 Polyglot

Polyglot [13] is an extensible frontend for Java that performs all the semantic checks required by the language. It is structured as a list of passes that rewrite an AST, and build auxilliary structures such as a symbol table and type system.

The extensibility of Polyglot is achieved in a number of ways. Polyglot allows a grammar to be specified as an incremental set of modifications to the existing Java grammar, and the tree rewriting portion can be extended without modifying the base compiler. New AST nodes may be added; they extend existing nodes and give definitions of the specific methods required by compiler passes that are relevant to them. New passes may be added between the existing passes. In addition, the behaviour of existing nodes in existing passes can be modified using *delegates* [13], achieving the same task in Java as intertype declarations do in AspectJ. Strict use of interfaces and factories throughout Polyglot makes it easy to modify structures such as the type system.

3.2 Soot

Soot [19] is a Java bytecode analysis toolkit based around the Jimple IR, a typed, three-address, stack-less code. Jimple is low-level enough for pointcut matching, in that the granularity of a join point is at least one entire Jimple statement.

It is high-level enough for weaving and easy analysis; in particular, during weaving, we need not worry about implicit operations on the computation stack, because all operations are expressed in terms of explicit variables.

Soot can produce Jimple from both bytecode and Java source code. The source frontend, JAVA2JIMPLE, makes use of Polyglot to build an AST and perform frontend checks, and then generates Jimple. As output, Soot generates Java bytecode. This process includes important optimisations for generating efficient bytecode [19]; these are necessary even for today's JITs. Soot also supports an annotation framework [16] which allows arbitrary tags to be attached to the code and automatically propagated through all transformations and all its intermediate representations. We make extensive use of tags to track information flowing through *abc*.

4 Architecture

In Section 2 we introduced the static and dynamic language features that must be handled by an AspectJ compiler, and in Section 3 we discussed our basic building blocks, Polyglot for building the frontend and Soot for building the backend. Of course, the big question is how to fit these building blocks together so that in the end one has a nicely structured AspectJ compiler that can handle both the static and dynamic features of AspectJ. In this section we address the design of the architecture, and then in Section 5 we focus on how to handle specific language features in more detail, where the implementation of some language features crosscuts several parts of architecture.

Figure 3 shows a high-level view of the *abc* architecture: the compiler takes .java and .class files as input, and produces woven .class files as output. An important point about AspectJ compilers is that the files given to it as explicit input are considered differently from classes that are found implicitly when the compiler must resolve classes from the class path. Classes corresponding to the explicit inputs are said to be *weavable*: aspects can weave into these classes, and it is the woven version of these classes that will be output by the compiler. Classes corresponding to the implicitly processed classes are not weavable.

As shown in Figure 3 we have split the architecture into four major components, two in the frontend and two in the backend. Compiler writers will immediately see that this architecture is different from the usual view of a compiler as a frontend and a backend connected via an intermediate representation.

The first major difference is that the frontend and backend of *abc* are connected via two data structures, the IR of the program (Java AST) and the AspectInfo data structure. The interesting point here is that in order to use standard Java compiler tools we must be able to tease apart the incoming AspectJ program into a standard Java part, represented as Java ASTs, and and aspect-specific part that captures all of the key information about aspects and how the aspects related to the Java IRs. This process is represented by the *Separator* box in Figure 3.

The second major difference between an AspectJ compiler and a standard Java compiler is that the backend must deal with both weaving the static AspectJ language features (static weaving), and weaving the dynamic language features (advice weaving). As shown in Figure 3 the static weaving is performed in conjunction with the code generation of the Jimple IR, and the advice weaving is performed on the Jimple IR.

In the remainder of this section we visit each of the four major components of the architecture, discussing the relevant details of each component.

4.1 Polyglot-based Frontend

We used Polyglot as the building block for our frontend. Polyglot allows us to define the AspectJ grammar in a separate definition file, as a natural extension to the Java grammar. It turns out that the exercise of specifying a complete LALR(1) AspectJ grammar had not been done before, and so this is another contribution of our project.[3]

A big challenge in developing our frontend was defining and implementing the semantic checks. AspectJ requires a large number of semantic checks in addition to the ones required by pure Java. Most of these checks have been implemented in Polyglot passes. Unlike in Java, where all semantic checks can be performed in the frontend, some semantic checks for AspectJ depend on the result of backend weaving, and thus some semantic checking has to be deferred until after weaving has occurred. As a result both exception checking (described in Section 4.4) and some checks related to advice weaving (described in Section 5.4) must be delayed until the backend.

[3]The *ajc* compiler uses a combination of an LALR(1) grammar and a hand-written top-down parser, so it does not provide a complete unified specification.

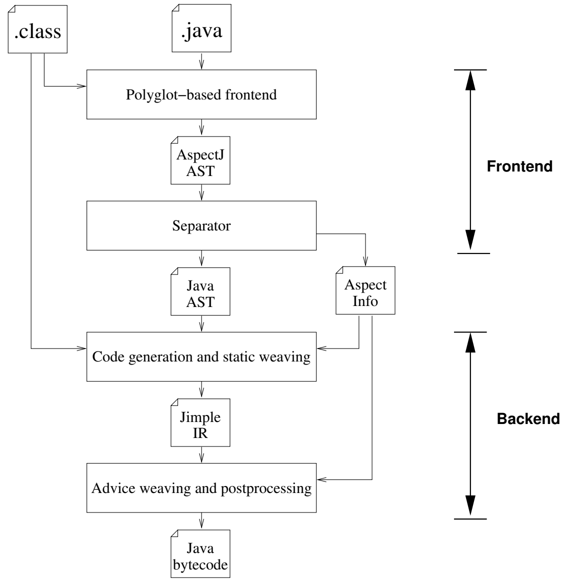


Figure 3: High-level overview of the components of the *abc* compiler

To implement the semantic checks, we only overrode 14 AST nodes of pure Java in minor ways; everything else was handled with new AST nodes and new visitor passes. The changes we made to the passes are summarised in Figure 4. This overview has been simplified for expository reasons; the actual number of passes for semantic checking in *abc* is 27, compared to 13 in the original Polyglot compiler. The number for *abc* is so large because we strove to minimise dependencies between passes, and therefore each new pass performs only one specific task.

The semantic checking of AspectJ source files depends on the static weaving, since, for instance, the code might refer to members introduced by intertype declarations. This makes the dependencies between passes quite subtle. In particular, checking *declare parents* needs the class hierarchy and inner-class relationships to be available, both for doing pattern matching (which depends on the hierarchy as described in Section 5.1) and for checking that the hierarchy introductions are valid. On the other hand, disambiguating the class names found in method signatures needs the final hierarchy in place, so this must happen after *declare parents*.

Similarly, to check the validity of intertype declarations, information about the existing class members must be available. Furthermore, anything that depends on the presence of class members (in particular the disambiguation of method bodies) must know about intertype members as well. Thus, semantic checking of intertype declarations must happen in connection with the pass that inserts the normal class members.

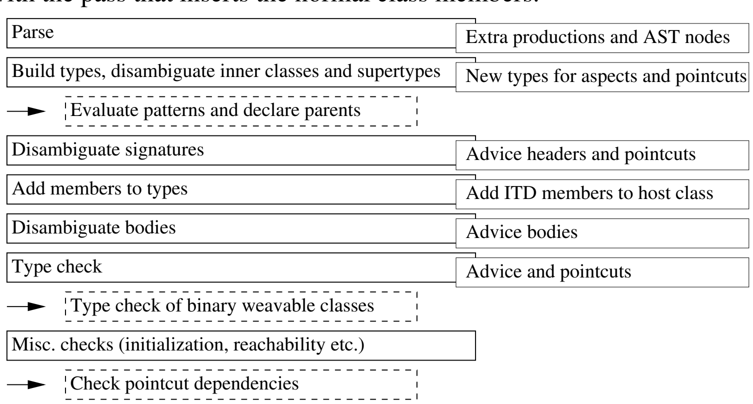


Figure 4: Simplified list of the compiler passes of Polyglot and how *abc* extends them. The solid boxes on the left show the original Polyglot passes for pure Java. On the right-hand side, in overlapping boxes, we have indicated which passes were changed. Finally, the dashed boxes with arrows indicate where we inserted new passes.

4.2 Separator

The key to our compiler architecture is the Separator, which splits the AspectJ AST (with associated type information) into a pure Java AST and the *AspectInfo* structure to record aspect-specific information. The *AspectInfo* includes all information that the backend needs from the Polyglot AST, so the backend does not use the AST at all, only the Jimple representation and the *AspectInfo*.

We now list the main components of the *AspectInfo* structure:

- All AspectJ-specific language constructs. For all constructs that contain Java code, the code is placed into placeholder methods in the Java AST, and the *AspectInfo* references these methods. It is important not to weave into some methods created by the compiler, so these are identified.
- An internal representation of the class hierarchy and inner class relationships.
- A list of weavable classes.
- Information about fields and methods whose names have been name mangled, or to which extra arguments have been added.
- A representation of types, classes and signatures that can be used throughout the whole compiler. This representation is independent of both Polyglot and Soot, and it provides a bridge for communicating type information between the two frameworks.
- Information about relative precedence between advice.

The separation process runs in roughly four steps, implemented as a number of Polyglot passes. The four steps of separation are:

1. **Name mangling**. The names of some intertype declarations must be mangled (see Section 5.3).
2. **Aspect methods**. Code from AspectJ constructs are inserted into pure Java methods, and dummy **proceed** methods are generated for proceed calls in **around** advice.
3. **Harvesting**. All AspectJ constructs are harvested from the AST and put into designated data structures in *AspectInfo*.
4. **Cleaning**. All AspectJ constructs are removed, leaving a pure Java AST. JAVA2JIMPLE sees aspects as plain Java classes containing the placeholder methods.

4.3 Code Generation and Static Weaving

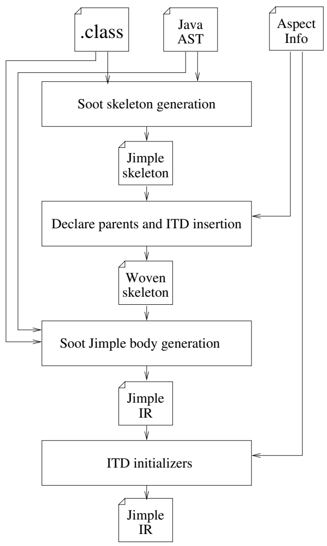
The AST passed to JAVA2JIMPLE might not correspond to a valid Java program in itself, since it may refer to members to be introduced by intertype declarations. Furthermore, it might depend on the class hierarchy being updated by *declare parents*. For these reasons, the translation from Java AST to Jimple code cannot happen as one atomic action.

To solve this problem, we take advantage of an existing feature of Soot. In Soot, the translation of both source and class files to Jimple happens in two stages: one to generate a skeleton, consisting of just the class hierarchy and the member structure of classes, but without any method bodies. The second stage generates the bodies in Jimple.

Figure 5 shows how the static weaving fits in between these two stages. After the skeleton generation, we adjust the hierarchy according to parent declarations and intertype declarations. The woven skeleton is then input into the Soot Jimple body generation. Finally, delegation code for intertype field initializers is generated.

4.4 Advice Weaving and Postprocessing

Once weaving of static features is complete and Jimple has been generated, we weave advice. The structure of advice weaver, and the final phases stages of the *abc* backend, is shown in Figure 6.



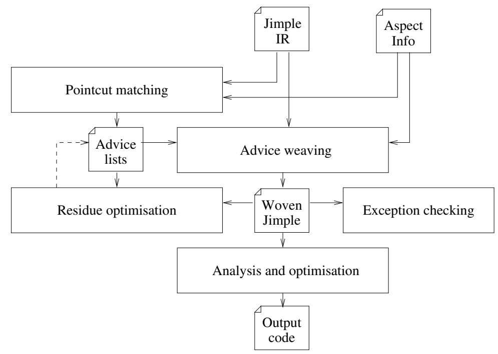


Figure 5: The two-step process of generating Jimple code while doing static weaving

Figure 6: The structure of the advice weaver and final stages of the *abc* backend

The process consists of two main steps, matching (see Section 5.4.1) and weaving (see Section 5.4.2). Matching determines the static locations (*shadows*) where each pointcut may match, and which dynamic checks are necessary to deteremine whether it matches. Weaving inserts the checks and the advice into the code.

At the same time as pointcut matching and advice weaving, we handle certain features that turn out to fit neatly into the same framework: *per* aspects (a construct for creating instances of an aspect), *declare soft* (for masking checked exceptions), *declare warning* and *declare error*. One side effect of implementing the *declare soft* construct is that we cannot verify that checked exceptions are declared correctly until we have dealt with this construct, since it has the effect of converting checked exceptions into unchecked exceptions. As a result, exception checking is carried out after the advice weaving process, rather than in the frontend as would be normal for a Java compiler.

Since one major goal of *abc* is to implement AspectJ features as efficiently as possible, we make it possible to perform analyses on the woven code, and use the analysis results in the weaving process to produce improved code. To support this, *abc* supports *reweaving*. Weaving is first performed, the analyses run on the woven code, weaving is undone, and then redone making use of the analysis results. The whole process can be repeated if desired.

Finally, *abc* runs a number of standard Soot optimisations, such as copy propagation and dead code elimination. Some of these are extended to add special knowledge of the *abc* runtime library; for example, the intraprocedural nullness analysis is extended to exploit the fact that certain factory methods in the *abc* runtime library never return *null*.

5 Implementing Language Features

In the previous section, we have described *abc* by giving its general architecture and points of interest about each of its components. We now adopt a different viewpoint, and show how various AspectJ language features are implemented within this architecture. The features that we focus on here are: implementing AspectJ patterns (*name matching*), the *declare parents* construct, *intertype declarations* and finally how the weaving of *advice* is implemented.

5.1 Name Matching

Many AspectJ constructs use patterns to pick out specific classes or methods to act on. The basic component of these is the name pattern; this selects classes textually by name. For instance, to select all classes in a package named *ast* that need support for break labels in a compiler, you might write *ast.*Loop* || *ast.If* || *ast.Switch*. This would match, amongs others, a class named *ast.WhileLoop*.

Finding the set of classes matched by a name pattern corresponds to normal Java name lookup. It follows the same scope rules, only it looks for all names matching a pattern, rather than a single name. To avoid performing this lookup process every time the name pattern is queried (which can happen many times), these matching sets are explicitly calculated for each name pattern before they are needed by any matching operations.

Name patterns range over all classes in the class path. However, all uses of patterns can be reduced to two cases: ranging over all weavable classes (this is the case for *declare parents*, for example), and ranging over all classes referred to in the program (this is used to match method patterns, among other things).

All pattern matches performed in the frontend range over the former domain (weavable classes). All class declarations in the AST and class files are collected for this purpose. After the *declare parents* pass, name patterns must be re-evaluated, in the updated class hierarchy. Finally, patterns need to be evaluated yet again after Soot has loaded all the classes referred to in the program, for use in the pointcut matcher in the backend.

5.2 Declare Parents

The *declare parents* construct allows an aspect to inject classes into the inheritance hierarchy, and to make classes implement additional interfaces. Figure 2(a) demonstrates a very simple use of *declare parents*.

The validity of a *declare parents* declaration involves some constraints on the class hierarchy (classes can only be inserted into the hierarchy chain, not completely replace the parent classes), plus some structural requirements on the child class (must actually implement the methods of the interface, must contain appropriate constructor calls etc.). All of these must be checked in the frontend.

The hierarchical constraints are checked in the *declare parents* Polyglot pass itself. Care must be taken here, as the validity of *declare parents* declarations might depend on the order in which different declarations (or even different classes matched by the same declaration) are handled. Handling the child classes in topological order, starting with `Object`, ensures that a unique valid interpretation is found if one exists.

For child classes from source, the structural requirements are taken care of by the normal Java checks, since these take place after the *declare parents* pass. For classes from class files, the checks must be performed explicitly.

All checks are performed in the frontend; the weaver for *declare parents* then modifies the hierarchy in Soot. Additionally, when a new superclass has been set on a class read from a class file, all superclass constructor calls must be changed to call constructors in the new parent, as these calls are represented as `invokespecial` instructions with the old parent class as explicit receiver class.

5.3 Intertype Declarations

When implementing intertype declarations, the main task is to make Polyglot's type-checker aware of the new members that are introduced by aspects. Polyglot includes a pass called ADDMEMBERS that populates class types with their members. Intertype declarations add their own type to the host class type during this pass. Note that this is *not* the same as actual weaving: we manipulate types only, not ASTs. The weaving of intertype declarations happens much later, in the static weaver.

Visibility A complication is introduced by the fact that visibility is always interpreted from the originating aspect. So for example, if we have two aspects *A* and *B*, and both contain a declaration *private int C.f*, then there are in fact two fields introduced in *C*, and they are only visible from their origin. To cope with this, we introduced subclasses of the AST nodes for class members (constructors, fields and methods) that are used for intertype-declared members and keep track of the origin of an intertype declaration. Furthermore, the accessibility test in Polyglot was overridden so that it uses that origin instead of the host class of an intertype declaration.

Scope inside intertype declarations The visibility rules are similarly applied to resolve variable and method references inside intertype declarations. The environment for an intertype method *C.foo()* in an aspect *A* is built up as follows: first, we have everything that is in scope inside *C* and which is visible from *A*. Next, we have the scope of *A*. Note, however, that it is an error to refer to instance variables of the aspect: as far as the aspect is concerned, the body

```
public class A {
  int x^1;
  class B {
    int x^2;
  }
}
```

```
aspect Aspect {
  static int x^3;
  static int y^4;
  int A.B.foo() {
    class C {
      int x^5 = 3;
      int bar() { return x^5 + A.this.x^1;}
    }
    return this.x^2 + (new C()).bar() + y^4;
  }
}
```

Figure 7: Scope rules for intertype methods.

of *foo* is a static context. The AspectJ rules for one intertype declaration overriding another are somewhat complex, and omitted for reasons of space.

This environment (consisting of the visible scope of the host class followed by the aspect) is used to disambiguate uses of *this* and *super* that may occur in the body of *foo*: we have to distinguish whether they refer to the host class *C*, to some local class, or to an aspect. Such disambiguation must also be applied to references that have an implicit *this* receiver. The example in Figure 7 illustrates this: each field has been labelled with a superscript to link declarations and references.

Because Polyglot is based on the rewriting paradigm, it is easy to implement these rules by introducing appropriate new AST nodes for *this* and *super* in the host class. Furthermore, by subclassing the type of environments, we can keep the necessary information about intertype declarations to decide for each variable whether it refers to the host class or not.

Mangling The visibility rules also imply that names of non-public intertype declarations must be mangled prior to code generation: a private ITD becomes a public member of the host class, but only the originating aspect should know its name. A subtle issue is that sometimes the mangling between several entities must be coordinated. For example, let *A* be an abstract class and *B* a concrete class that extends *A*. Now if we introduce a package-visible abstract method *foo* into *A*, and an implementation of *foo* into *B*, both must be mangled to the same name. For this purpose, we introduced a new pass that computes equivalence classes of intertype declarations that must get the same name. A subsequent pass then carries out the name mangling, renaming both declarations and references.

In Polyglot, this is nicely implemented by storing the relevant information (about equivalence classes and mangled names) inside the type for the intertype declaration. It is then very easy to fix up the references as required.

AspectInfo and code generation Our implementation strategy leaves the code for intertype methods as static methods in the originating aspects. There are two reasons for this decision. First, there is no need to generate accessor methods for accessing members of the aspect scope (and that is the vantage point for visibility tests). Second, the weaver will correctly use the aspect as the lexical scope for matching *within* pointcuts. To illustrate, we return to the *AddValue* example of Section 2. After the ASPECTMETHODS pass, the code for *getValue* in the *AddValue* class will be:

```
public static getValue$4(final node.Node this$6) {
  return this$6.AddValue$value$3;
}
```

This is then called by a delegating method in *Node* that passes the **this** pointer as an argument. Sometimes there is still a need to generate accessor methods, for example if the host class is nested, and there is a reference to an enclosing class in the intertype method. Accessor methods are also necessary for the implementation of *privileged* aspects, which by definition are able to override all the visibility rules and can access any members, of any class in the system. Due to space constraints, we omit a detailed discussion.

5.4 Advice

A piece of advice consists of the pointcut specifying when it should apply, together with some code to be run. The frontend of *abc* constructs a method body with a synthetic name to hold this code, and places the pointcut and the name of this method in the *AspectInfo* structure. The job of the backend is then to find the static locations in the code where each pointcut might match (the join point shadows), and to insert code that will check at runtime whether or not the pointcut does actually match, and call the method implementing the advice body with the appropriate parameters.

As well as advice that is defined directly in the user's aspects, various forms of synthetic advice are used to implement features of the AspectJ language such as *cflow* pointcuts, *declare soft*, and aspects that are only instantiated conditionally (*perthis* etc). We return to this point after explaining the mechanics of how normal advice is inserted.

In *abc*, finding where advice might apply (*matching*) and inserting calls to that advice (*weaving*) are done in two distinct phases; the matcher produces a list of "advice applications" that is then passed to the weaver. We did this (rather than immediately inserting code as advice is found to apply) for two reasons.

Firstly, there are specific rules of *precedence* stating in which order multiple pieces of advice applying at the same join point should run, and it is most convenient to weave advice in order of precedence. Unfortunately we cannot simply sort the complete list of advice before matching, because it is legal to have a cycle in the precedence relationship, so long as that cycle is not actually realised at any particular join point shadow. Having an intermediate list that we can sort before weaving is therefore helpful.

Secondly, as we mentioned in Section 4.4, we want to support *reweaving* to produce better runtime code using analysis results from a first attempt at weaving. Again, the presence of an explicit intermediate list makes this process easier.

5.4.1 Matching

Pointcuts can only match at specific *join points* during the program's execution. Each join point corresponds to a static *join point shadow* in the program. The pointcut matcher first identifies all the join point shadows in the program. For each shadow, it tests each pointcut to see if it could possibly match at that point.

Figure 8(a) shows an example of some Java code and a pointcut. The `mainEval()` pointcut from the `CountEvalAllocs` aspect picks out all join points within the `Main` class where `eval()` is called, and so in particular the call from within the `run()` method is a join point shadow at which the *before* advice in this aspect can apply.

```
public class Main {
   ...
   public void run()
       { eval() }
   ...
}
public aspect CountEvalAllocs {
   ...
   pointcut mainEval() :
      call(* *.eval(..)) && within(*.Main);

   before () : mainEval()
      { allocs = 0; }
   ...
}
```

(a) Source Java and AspectJ code

```
public class Main {
   ...
   public void run() {
      Main this;
      CountEvalAllocs theAspect;

      this := @this: Main; // give arg 0 a name
      nop // beginning nop for shadow
      // get the singleton aspect instance
      theAspect = staticinvoke <CountEvalAllocs:
         CountEvalAllocs aspectOf()>();
      // run the before advice
      virtualinvoke theAspect.<CountEvalAllocs:
         void before$0()>();
      nop // jump here if residue fails
      // run the original code at the shadow
      virtualinvoke this.<Eval_Main: eval()>();
      nop // ending nop for shadow
      return;
   }
   ...
}
```

(b) Woven Jimple

Figure 8: An example of matching and weaving

Regularised pointcut language The problem of checking whether a particular pointcut applies at a given shadow naturally splits itself into three parts: A shadow occurs inside a method body, which is itself contained within a class; shadows also have a specific type, for example one might cover the entire execution of the method, or a single instruction that sets a particular field. A pointcut can place restrictions on any of the containing class, the containing method, and the shadow type.

As it turns out, some primitive pointcuts in the AspectJ language place restrictions on more than one of these parts, and in addition there is a significant amount of duplication and overlap between different pointcuts. For example, the pointcut *execution(int foo())* specifies that we are only interested in execution join points inside methods with the given signature, while the pointcut *withincode(int foo())* also specifies that the join point should be inside such a method, but imposes no restriction on its type.

As a result, we have chosen to make the implementation simpler by working with a modified pointcut language in the backend, and having the frontend translate pointcuts into this modified form when constructing the *AspectInfo* structure. In our modified language, each of the primitive pointcuts restricts at most one of the three parts mentioned above; *i.e.* it either specifies the containing class, the containing method, or the join point type.

This regularised language also partitions certain AspectJ pointcuts into two different pointcuts; for example *withincode(...)* can take either a method signature or a constructor signature as an argument, but in our backend language there are two pointcuts, *withinmethod(...)* and *withinconstructor(...)*.

Therefore, the pointcut *execution(int foo())* will be translated into *withinmethod(int foo()) && execution()*. In the regularised language, the latter conjunct is only a restriction on the join point type and does not specify anything about the containing method.

Dynamic residues Once the matcher has identified that a pointcut might apply at a join point shadow, it remains to generate some runtime code for that shadow to determine whether the pointcut does actually apply each time an associated join point occurs (*i.e.* the control flow of the program reaches that shadow). In some cases, we will statically know that the pointcut will always apply at the shadow, so the corresponding advice body will be executed unconditionally.

As well as deciding whether an advice body should execute at all, it is necessary to gather certain values before calling it. All advice bodies run as instance methods in the aspect that defines them, and it is necessary to call the static `aspectOf` method in that aspect to obtain an instance for use as the receiver of the advice call. We can see an example of this call in the woven code in Figure 8(b). The `aspectOf` method itself is automatically generated in an aspect body when compiling it into a class.

There are a number of features of the pointcut language which require runtime checks or the passing of values. Most important of these are the *this*, *target* and *args* pointcuts, which expose, where they exist, the value of the current object instance, the receiver at the join point, and the arguments being passed at the join point. Each of these can be either given a variable name as an argument, in which case the relevant value will be available in the advice body under this variable name, or a type, in which case a check will be made at runtime that the value has the appropriate type unless this can be statically determined. (In fact, variables in pointcuts must be declared with their types, and so a type check is also carried out when a variable name is specified).

It is the role of the matcher to establish what checks need to be done at runtime and what information needs to be gathered, but as described above it does not actually add runtime code. Therefore, it records this information in a structure known as a *dynamic residue*, which the weaver later processes.

5.4.2 Weaving

The role of the advice weaver is to actually generate the runtime code for running advice bodies where appropriate. We use the facilities provided by Soot to make this process as simple as possible. For example, the Soot backend carries out optimisations such as removing *nop* instructions and dead code, so our code generation strategy does not worry about leaving these in the code it outputs, which makes its design significantly simpler. In Figure 8(b), we see the results of weaving before these optimisations are applied.

Another property of Soot that helps the design of the advice weaver is that since Jimple is a three-address code with explicit variable names rather than implicit stack locations, we can simply refer to a variable at the place it is needed, rather than having to make sure that its value is available on the stack. This is particularly useful when passing values to advice bodies.

Preparing join point shadows One important problem is that we need to ensure that multiple pieces of advice applying at the same join point are run in the correct order. In particular, *after throwing* advice, a specific form of *after* advice which only runs if an exception is thrown at the join point, needs careful treatment to ensure that it interacts correctly with the existing exception behaviour of the join point and of other advice applying at it. We also need to make sure that jumps are fixed up correctly; statements that branch to the beginning of a join point shadow should now branch to the first piece of advice that might run at that shadow (it is not possible for an existing statement to branch to the middle of a shadow).

Our approach is to first insert *nop* statements at the beginning and end of each shadow, and then to weave advice in an "inside-out" order — that is, *before* advice that should run "closest" to the original code of the join point is woven first. The idea is that at each stage, the *nop* statements enclose the entire join point including advice that has been inserted so far, and that the next piece of advice to be woven is inserted just inside the *nop* statements — immediately after the beginning one for *before* advice, and immediately before the ending one for *after* advice. This keeps the weaving process as simple and as modular as possible — the procedure for inserting the *nop* statements takes care to ensure that jumps and exception handling ranges are correctly modified, and the subsequent weaving process can largely ignore this. For example, if an exception range covers the original code at the shadow, it should cover the entire join point after weaving, but if it has been introduced by *after throwing* advice, it should only cover the original code and any advice that was woven before the *after throwing* advice; advice that is woven afterwards should not be within the exception range. The *nop* statements allow us to tell the difference, because in the former case they will be included in the exception range, but in the latter case they will not.

An added complication is that certain types of join point shadows do not fit nicely into the single-entry single-exit (ignoring exceptions) model implied by the above approach. For example, an execution join point might terminate at any one of a number of *return* statements. Therefore, we first transform the code where necessary, replacing these *return* statements with jumps to a single *return* at the end of the body, first storing the value to be returned in a local variable if necessary.

Similarly, the *preinitialisation* and *initialisation* join points can span multiple constructors, if one constructor calls another in the same class using *this(...)*. We therefore inline such calls to ensure that the code for each shadow is fully contained within a single method.

Inserting advice Each type of advice (*before*, *after* and *around*) has its own weaver, which inserts code in the appropriate position of the join point shadow. As mentioned before, *before* advice goes immediately after the beginning *nop* of the shadow (an example of this can be seen in Figure 8(b)), and all forms of *after* advice go immediately before the ending one. A novel strategy, described in [10], is used for *around* advice; the key detail for the purposes of this paper is that it lifts all the code found between the two *nop* statements at the time of weaving into a separate method, replacing it with code to implement the advice, which can itself call back to the original code.

In fact, it is only *after returning* advice, another specific form of *after* advice that only runs on normal termination of a join point, that needs to be placed at the end of the shadow; *after throwing* advice is implemented by an exception handler which ends by rethrowing the original exception, so can be placed anywhere in the method. For simplicity, we choose to also place it at the end of the shadow. "Full" *after* advice, which runs both after normal termination of the join point and when an exception is thrown, is actually implemented by weaving both *after returning* and *after throwing* advice.

Once we have identified where the advice should go, the next step is to weave code for the dynamic residue. We assume that any dynamic residue could fail; this may leave some dead code around in the case of residues that cannot, but this is tidied up later by the Soot backend. Thus, each dynamic residue is woven with two exit points; one which runs the advice body and one which skips it. In Figure 8(b), the *nop* labelled as "Jump here if residue fails" is the exit point point for failure (which is never jumped to in this example), and the call to the advice body immediately after is the exit point for success.

5.4.3 Synthetic advice

Certain constructs in the AspectJ language other than advice have pointcuts associated with them, and require code to be run at the join points picked out by these pointcuts. For example, users of *declare soft* specify a pointcut where certain exceptions should be softened, which requires inserting code at the relevant join point shadows to catch the exception, wrap it up as a *SoftException* and throw this new exception.

Of course, this is very similar to what is required to implement advice declarations; the main difference is merely that the code to be inserted is not a call to an advice body. It is natural to use the same implementation strategy for such constructs, and indeed the frontend of *abc* transforms them into "synthetic" advice declarations to be processed along with the normal pieces of advice.

The final constructs that the advice weaver deals with are *declare warning* and *declare error*. These also specify pointcuts, but no code is inserted at the relevant join points; they merely cause the compiler to emit warnings or errors if any such join points are found. Since they must be evaluated at compile-time, it is an error to specify a pointcut which would require runtime code to check whether it applied or not. In *abc* these constructs are also treated as synthetic advice declarations, but instead of generating a dynamic residue for the code weaving phase, a warning or error is emitted as appropriate.

6 Related work

ajc is the original compiler for the AspectJ language, and was written by the language's designers. It builds on the Eclipse Java compiler, while the backend makes use of a customised version of BCEL. The separation from the Eclipse compiler is however not complete, and a painful merge has to be undertaken when the base compiler is upgraded. Implementing a weaver with BCEL is hard by comparison with Soot; a detailed description of the weaver in *ajc* can be found in [7]. In summary, the structure of *abc* is similar to that of *ajc*, separating the pure Java and aspect-specific information, and leveraging existing frontend and backend technology. However, *abc* achieves a complete separation from these building blocks, using them without any modification.

The general strategy of weaving dynamic features in AspectJ, leaving dynamic residues where needed, is nicely explained in terms of partial evaluation in [12]. AspectJ is by no means the only aspect-oriented language, however, and in the remainder of this section we give a quick overview of the most important alternatives and their implementation strategy.

AspectC++ is an extension of C++ with aspects, which provides pointcuts and advice, but there is no support for advanced static weaving features such as *declare parents* [8]. It is implemented as a source-to-source transformer. As explained earlier, we believe much is to be gained from weaving on an appropriate intermediate representation - not only the ability to weave binaries, but also to simplify the implementation of the weaver.

AspectWerkz is a framework for the application of aspects to Java programs. The instructions to the weaver can be given in a variety of meta-notations, including XML and Java 1.5 attributes. The AspectWerkz framework is of a highly dynamic nature, allowing aspects to be enabled and disabled at run-time. This is achieved via a mechanism akin to the observer pattern: each piece of advice becomes a kind of listener, while joinpoints generate events to notify the advice. In his paper on the implementation of AspectWerkz [4], Jonas Bonér claims the overheads are negligible. To assess that claim, we translated a few benchmarks from [6] into AspectWerkz, in particular a variant of *Figure* and of *NullCheck*. We found that the code produced by AspectWerkz for *Figure* runs 1000% slower than that produced by *abc*, and *NullCheck* runs 600% slower — even when using the *offline weaving* feature of AspectWerkz, which performs weaving at compile-time instead of load-time. Similar observer-style implementation techniques are employed in Eos (an aspect-oriented extension to C#) [17] and JAC (a framework for distributed aspect-oriented programming) [15]. AspectWerkz aims for load-time weaving, and thus the efficiency of its weaver needs to be balanced with the efficiency of the generated code.

JBoss AOP is an aspect oriented framework similar to AspectWerkz, but it is more targeted towards the JBoss Application Server. The main implementation technique is a framework called Javassist [5] for writing bytecode translators. Javassist has been carefully honed to produce efficient translators, again with a view towards load-time weaving. By contrast, our use of Soot was motivated by the desire to produce efficient object code, while the time taken by the weaver itself is less important.

Neither AspectWerkz nor JBoss AOP appears to implement the level of static checking afforded to us by the use of Polyglot: again this is motivated by the desire to produce efficient translators. Indeed, AspectWerkz lacks certain features of AspectJ that require more transformation or checking than others. In particular it lacks initialisation joinpoints, exception softening, precedence declarations and parents declarations. It also lacks the ability to issue compile-time warnings and errors based on pointcut matching.

7 Conclusions and Future Work

In this paper we have presented how we designed and implemented the *abc* AspectJ compiler, building upon two existing compiler toolkits, Polyglot and Soot. The *abc* compiler is a complete implementation of the AspectJ language, which can be used as an alternative compiler for AspectJ applications, or as a workbench for language extensions and compiler optimisations.

There were two main contributions in this paper. First, we demonstrated how to build the architecture of *abc* around the Polyglot and Soot building blocks. It was a non-trivial exercise to make these building blocks fit together, but with the correct design of the *AspectInfo* data structure we showed how the AspectJ-specific information could be cleanly separated from the pure Java part, thus enabling us to use Polyglot and Soot as Java tools.

We found that there were distinct benefits of building upon such powerful tools. We used Polyglot's extensible grammar system to specify AspectJ as a clear extension of Java, and Polyglot's pass mechanism to insert new passes relevant to AspectJ. We also used Polyglot's extension mechanisms to implement the relatively complex semantic checks required for AspectJ, particularly as they related to AspectJ's *intertype declarations*, which have quite complex semantics. We found a large benefit from using Soot as our backend, mostly due to the use of Soot's Jimple intermediate form, but also because Soot easily handles inputs as either class files or Java source files. By basing our matcher and weaver on Jimple we found that it was quite easy to specify matching rules and also quite straightforward to actually perform the weaving. Finally, Soot's built-in optimizations allow us to produce code that has been cleaned-up after weaving, and gives us the opportunity to implement AspectJ-specific optimisations in the future.

Our second main contribution was to show, in some detail, how we implemented the aspect-specific parts of our compiler, in particular how we handle name matching, the *declare parents* construct, intertype declarations and advice matching and weaving. For *declare parents* the main challenge was to fit its handling in the right position among existing passes: just enough information has to be available to do a first evaluation of the relevant patterns, but no other processing should be done yet. Regarding intertype declarations, our main obstacle was to determine and implement the correct scope rules. In fact, clarifying these scope rules has had an immediate impact on *ajc*. Finally, for advice and pointcut matching a salient point was the design of an intermediate representation for pointcuts that simplified the implementation. Perhaps the most promising part of our architecture, however, is the ability to weave, analyse the result, and weave again — this opens the way towards sophisticated analyses of AspectJ programs, for instance to implement the *cflow* optimisation proposed in [18].

The abc group found the project of building the compiler to be exceptionally fun, challenging and educational. We hope that others will learn from our experiences and that *abc* will continue to be a research platform for further work on compiling aspect-oriented languages. Our group is actively pursuing optimisation opportunities, and also new language extensions that require more sophisticated static analyses.

Acknowledgments

This work was supported, in part, by NSERC in Canada and EPSRC in the United Kingdom.

References

[1] abc. The AspectBench Compiler. Home page with downloads, FAQ, documentation, support mailing lists, and bug database. Available from URL: `http://aspectbench.org`.

[2] Eclipse AspectJ. The AspectJ Eclipse Project. Available from URL: `http://eclipse.org/aspectj`.

[3] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. Technical Report abc-2004-1, Available from URL: `http://aspectbench.org/techreports`, 2004.

[4] Jonas Bonér. Aspectwerkz — dynamic AOP for java. Available from URL: `http://codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf`, 2004.

[5] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *2nd International coference on Generative Programming and Component Engineering (GPCE '03)*, volume 2830 of *Springer Lecture Notes in Computer Science*, pages 364–376, 2003.

[6] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 149–168. ACM Press, 2003.

[7] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In K. Lieberherr, editor, *Aspect-oriented Software Development (AOSD 2004)*. ACM Press, 2004.

[8] AspectC++ Home. The AspectC++ Home Page. Available from URL: `http://www.aspectc.org`, 2004.

[9] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *European Conference on Object-oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[10] Sascha Kuzins. Efficient implementation of around-advice for the aspectbench compiler. M.Sc. thesis, Oxford University, 2004. Available from URL: `http://aspectbench.org/theses`.

[11] Ramnivas Laddad. *AspectJ in Action*. Manning, 2003.

[12] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Compiler Construction*, volume 2622 of *Springer Lecture Notes in Computer Science*, pages 46–60, 2003.

[13] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152, 2003.

[14] PARC. AspectJ PARC Archive. Available from URL: `http://www.parc.com/research/csl/projects/aspectj/`.

[15] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. JAC: An aspect-based distributed dynamic framework. *Software: Practise and Experience (SPE)*, 34(12):1119–1148, October 2004.

[16] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing Java using attributes. In *Compiler Construction, 10th International Conference (CC 2001)*, volume 2027 of *LNCS*, pages 334–554, 2001.

[17] Hridesh Rajan and Kevin J. Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC / SIGSOFT Foundations of Software Engineering*, pages 291–306, 2003.

[18] Damien Sereni and Oege de Moor. Static analysis of aspects. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 30–39, 2003.

[19] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.