The abc Group

Building the *abc* AspectJ compiler with Polyglot and Soot

abc Technical Report No. abc-2004-4

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

December 20, 2004

Contents

List of Figures

Abstract

Aspect-oriented programming and the development of aspect-oriented languages are rapidly gaining momentum, and the advent of this new kind of programming language provides interesting challenges for compiler developers, 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.

In this paper we 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. We provide a software architecture which cleanly separates these existing tools from the aspect-specific parts of the compiler.

A second important contribution of the paper is that we describe our implementation strategies for 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.

Finally, we found that in developing *abc* we clarified many language issues which in turn simplified the implementation.

Our *abc* compiler implements the full AspectJ language as defined by *ajc* 1.2 and is freely available under the GNU LGPL.

1 Introduction

Aspect-oriented programming (AOP) is rapidly gaining popularity and AspectJ [10] 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 [15] and currently developed and maintained as part of the Eclipse AspectJ project [2].

This paper describes the design and implementation of a new compiler for AspectJ, the *AspectBench Compiler*, *abc* [1], which is intended as a workbench for researchers interested in AOP language design and implementation.[1] Whereas the development of *ajc* has focused on integration with the Eclipse framework and on incremental and fast compilation, our motivation and design goals were quite different. Our original goals and resulting contributions can be summarised as follows:

Clearly defined and articulated compiler architecture for an AOP language: The overall architecture of a compiler for mainstream programming languages is very well understood and documented in numerous textbooks. However, AOP languages provide new challenges for compiler writers and the architecture of a compiler for an AOP language must reflect those challenges. Although the basic structure of the compiler as a frontend and backend remains, there are important differences. Our main contribution is a systematic description of AOP-specific issues in compiler construction.

For example, while in a standard compiler the frontend and backend need only communicate through an intermediate representation and symbol table, in an AOP compiler detailed aspect-related information must be transmitted. In addition, several of the components of such a compiler (*eg.* the name matcher, pointcut matcher, intertype declaration weaver and advice weaver) have no equivalent in a traditional compiler. Finally, some compilation phases, semantic checking for one, are made substantially more complicated, and new phases are introduced both to the frontend and backend.

Support for language extensions and optimisation of generated code: Our *abc* compiler is intended for use in research, and as such must be able to handle both AOP language research and compiler research. To that end, our design allows researchers to simply implement new language extensions and to implement new compiler analyses and optimisations (indeed, *abc* is already being used in this way).

Use of existing tools without modification: 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 [14] with the Soot framework for analysis and optimisation of Java [19].

[1] Readers who are not familiar with AOP languages such as AspectJ may wish to read Section 2 before proceeding with the rest of this introduction.

A substantial part of the design of *abc*'s architecture 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. Ours is the first AspectJ compiler to achieve a clean separation of the implementation of aspect-oriented features from these underlying tools (in particular, these tools have not been modified for use in *abc*). An important consequence of cleanly separating the AspectJ-specific parts of the implementation from the Java tools is that this separation provides a clear specification of how AspectJ extends Java.

Finally, a crucial goal in the design of *abc* was to apply Soot's existing frameworks for sophisticated analyses to aspect-specific optimisations. As these analyses work on pure Java intermediate representations, the design allows for weaving (producing pure Java), followed by analysis. The weaving process can then be undone so that a better weave, using analysis results, can be applied. This architecture for an optimising AOP compiler is another novel result of the development of *abc*.

In building the compiler, we also made some further contributions concerning the clarification of the language:

Clarification of the AspectJ language definition: When we started the *abc* project we searched for all relevant literature defining the AspectJ language. We found that the only complete language specification of the AspectJ language was the *ajc* compiler itself and its associated test suite. Thus, as we designed and implemented *abc* we had to reverse engineer much of the language specification.

This need to reverse engineer was evident even at the grammar level. The *ajc* grammar is a combination of a modified LALR(1) Java grammar specification and a hand-coded top-down parser. Thus, one of our first contributions was to develop a complete LALR(1) grammar specification using Polyglot's grammar extension mechanism to cleanly separate the Java part of the grammar from the AspectJ-specific part. This grammar is part of the *abc* distribution.

We also found several other places where the language specification needed to be clarified, for example the scope rules for intertype declarations, the precise meaning of the *declare parents* construct, the scope of name matching and the rules for matching pointcut expressions with alternative bindings. These clarifications have also helped improve the *ajc* compiler as recent releases of *ajc* incorporate many of the clarifications pioneered in *abc*.

Finally, language clarification has suggested improved implementation strategies. As an example, we noted that the AspectJ pointcut language is not as cleanly factored as it could be (perhaps unsurprisingly, as it developed over time). We have developed a regularised pointcut language that better separates orthogonal concerns. This has led to a simpler specification and implementation (in *abc*, weaving is done for the regularised language, and AspectJ pointcuts are simply translated into this form).

The structure of this paper is as follows. In Section 2 we provide a brief introduction to the most relevant features of AspectJ.[2] 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. In Section 6 we provide a comparison between the *ajc* compiler and our *abc* compiler. Finally, Section 7 reviews related work and Section 8 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 example aspects. As illustrated in Figure 1(a), most of the interpreter was generated using the SableCC parser

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

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

```
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

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

Declare Warning. The `StyleChecker` aspect in Figure 2 illustrates an interesting use of AspectJ, the *declare warning* construct[3]. This construct allows the programmer to specify a pattern, known as a *pointcut*, and a warning string. For each place in the program matching the pointcut, a compile-time warning is issued, using the string as the warning message. In our example we have specified a pointcut that matches all places where a field is set, and which are not within a method whose name starts with "*set*". In fact, the pointcut 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;
        ^-------^
```

Declare Parents and Intertype Declarations. 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 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, such as getting/setting fields, calling methods/constructors and executing method/constructor/initialiser 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 pointcut to match certain events, and then advice (extra code) to execute *before*, *after* or *around* the matching events. The pointcuts take the same form as those used with *declare warning*, but the language is slightly richer because they can depend on runtime events whereas those used with *declare warning* must be entirely statically evaluated. 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.

[3] There is also an analogous *declare error* construct.

Advice. 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. The second conjunct "**call**(*.**new**(..))" of this pointcut matches all constructor call events. The first conjunct restricts the pointcut to those events that occur within the dynamic scope of a call to `eval` (*i.e.* in the time span between the beginning and end of the call), using the *cflow* construct. This is of particular interest, as matching of *cflow* depends on runtime context, and in general runtime checks are necessary (this is also the case for other AspectJ pointcuts not covered here).

Around advice. The `ExtraParens` aspect contains a 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 pointcut 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.

The use of *proceed* can be quite complex — it can be executed many times or not at all, saved for later execution in a local class, and arguments can be changed. This makes *around* advice substantially more complicated than *before* and *after*, as it does not reduce to just injecting advice code, but can change the execution of existing code.

It should also be noted that our advice examples are very simple and do not illustrate all features of the language. In particular, advice may have *parameters* that are bound to runtime values in matching (this may involve runtime type checks). 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.* [12].

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 [14] 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 auxiliary 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* [14], 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 any 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 [17] 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 that are not explicitly input 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 an aspect-specific part that captures all of the key information about aspects and how the aspects relate 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 weaving, both the static language features (*static weaving*) and the dynamic 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.

The main issue in the design of the frontend is the large number of new semantic checks that AspectJ requires (in addition to those required by pure Java). In particular, the *declare parents* construct imposes restrictions on the class hierarchy and the affected children classes (see Section 5.2), while intertype declarations require the implementation of new scope and visibility rules (see Section 5.3). Furthermore, unlike in Java where all semantic checks are performed in the frontend, when compiling AspectJ programs some checks must be delayed until after weaving in the backend (in particular, exception checking, see Section 4.4).

Semantic checking is further complicated by the subtle dependencies between phases of static weaving and certain checks. For example, to check *declare parents*, the class hierarchy must be available, both for name matching (Section 5.1) and for checking validity of hierarchy introductions. However, disambiguation of class names in method

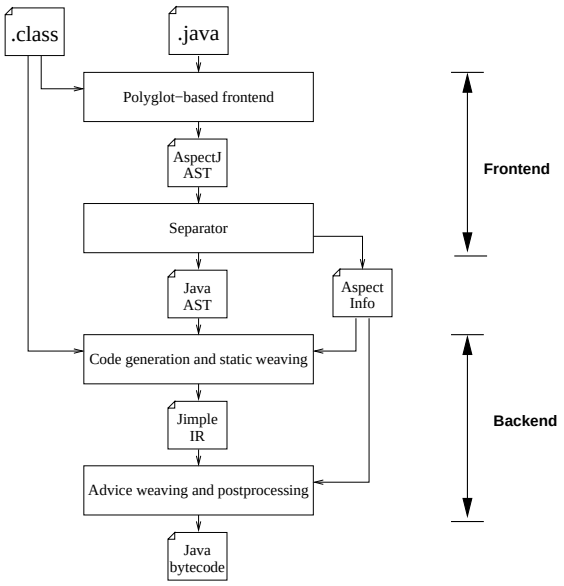


Figure 3: High-level overview of the components of the *abc* compiler

signatures requires the final hierarchy, and so must occur after *declare parents* instructions have been processed.

These complex checks and dependencies between them motivate the design of the frontend. The semantic checks were implemented by extending certain Polyglot passes, and adding some entirely new passes, in the order dictated by dependencies between the first phases of weaving and checks. The structure of the frontend is outlined below in Figure 4.

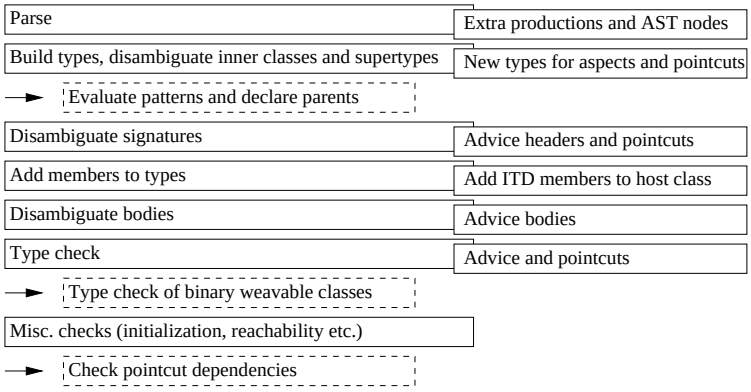


Figure 4: Simplified list of the compiler passes of Polyglot and how *abc* extends them. The solid boxes on the left show the original Polyglot passes for pure Java. On the right-hand side, in overlapping boxes, we have indicated which passes were changed. Finally, the dashed boxes with arrows indicate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

The design of the frontend was successful in separating AspectJ from the pure Java 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. In total, the number of passes for semantic checking in *abc* is 27, compared to 13 in the original Polyglot compiler. The large number of passes for *abc* is a result of the design goal of having each pass perform only one task to avoid creating further dependencies between 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 is 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 initialisers 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 the advice weaver and the final stages of the *abc* backend is shown in Figure 6.

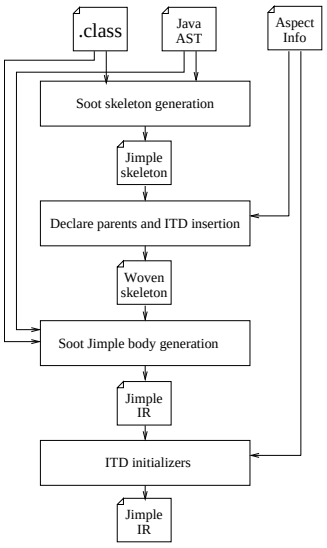


Figure 5: The two-step process of generating Jimple code while doing static weaving

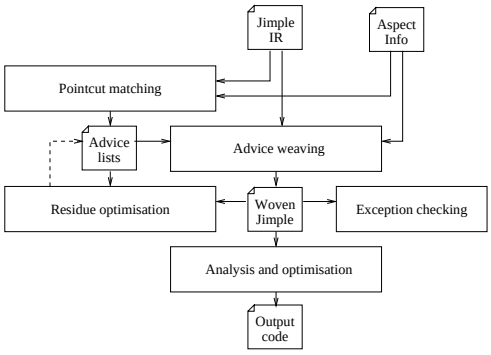


Figure 6: The structure of the advice weaver and final stages of the *abc* backend

The process consists of two main steps, matching and weaving (see Section 5.4). Matching determines the static locations (*shadows*) where each pointcut may match, and which dynamic checks are necessary to determine 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. This is supported through *reweaving*, illustrated in Figure 6. In reweaving, weaving is performed once naively to produce pure Jimple code. The resulting code is then analysed, as a pure Java program. The original weaving is then undone, and the aspects woven again, using information obtained in the analysis. This process can be iterated to improve precision. The upshot of this procedure is that aspect-specific analyses (for example, we have described an analysis for *cflow* [4]) can leverage existing Java analyses, such as call graph construction, without needing to make those aspect-aware.

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, among 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, but 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 need to be evaluated more than once during compilation, as they denote different sets of names in different contexts. 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 matches performed in the frontend range over the former domain (weavable classes), while patterns must be re-evaluated for use in the backend (after all classes referred to in the program have been loaded, and with the final hierarchy from *declare parents* in place).

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 challenge is that the type-checker must be aware of the new members that are introduced by aspects. This was a surprisingly difficult step in the development of *abc* and in the process of resolving this problem we defined some precise scope and visibility rules, described below. Thus, an important contribution of our work is the clear statement of these rules and their implementation, and careful consideration of all corner cases. Several improvements to *ajc* were prompted by this part of the development of *abc*.

Populating class types Polyglot includes a pass called ADDMEMBERS that populates class types with their members. In our *abc* extension, 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.

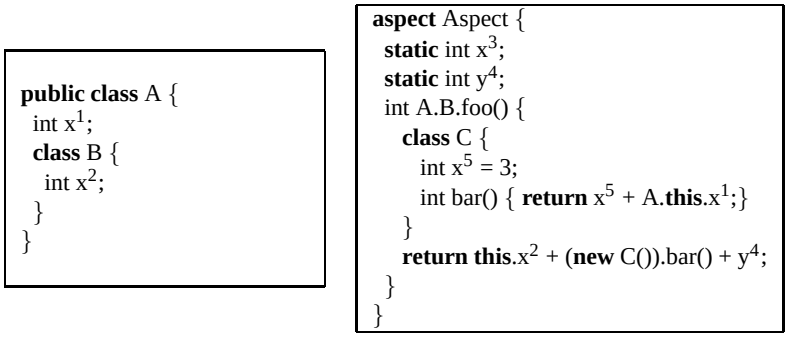


Figure 7: Scope rules for intertype methods.

Visibility A type checker for Java must consider both visibility and scope rules. When dealing with intertype declarations an extra 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. This is dealt with by identifying class members (constructors, fields and methods) that arise from ITDs as subclasses of the corresponding AST nodes that keep track of the origin of the ITD. The accessibility test in Polyglot was further overridden to use 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 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. This avoids the use of accessor methods for accessing members of the aspect scope (and that is the vantage point for visibility tests). Also, the method is then considered as part of the aspect for name

matching, which is the desired behaviour. 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.

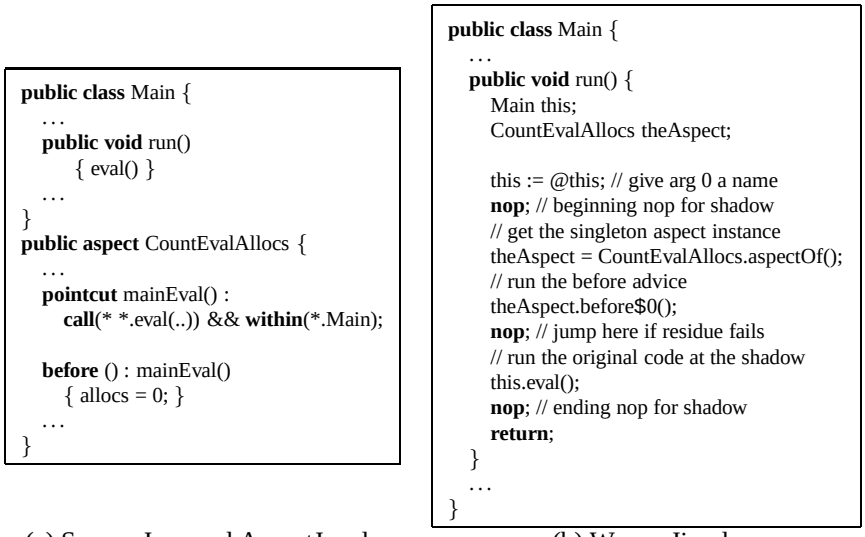
5.4.1 Matching

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.

Regularised pointcut language The problem of checking whether a particular pointcut applies at a given shadow is made more complicated by the fact that many AspectJ pointcuts check more than one property of a join point, and that there is a significant amount of overlap between pointcuts. For example, the pointcut *execution(int foo())* picks out join points based on two properties: their type (execution join points), and the methods their shadows occur in (only methods with signature *int foo()* are considered). This results in a substantial amount of duplicated work and makes the matcher unnecessarily complex if implemented in a straightforward manner.

As a result, we have defined a modified pointcut language in the backend that avoids this problem. Each pointcut in the regularised language checks exactly one of three properties of a join point: the type of the join point, the method that it occurs in, and the class containing this method.



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

(b) Woven Jimple

Figure 8: An example of matching and weaving

The frontend transforms AspectJ pointcuts into this regularised language for use in the backend. In doing so, certain pointcuts are transformed into a conjunction of more primitive pointcuts (for example, *execution(int foo())* becomes *withinmethod(int foo()) && execution()*, where each conjunct only checks one kind of property). As a further simplification, certain pointcuts are split into cases. An example of this is the AspectJ *withincode(...)* pointcut. This can pick out methods or constructors based on their signature; in our backend it is replaced by uses of the more specific pointcuts *withinmethod(...)* and *withinconstructor(...)*. The use of a simple, orthogonal pointcut language allows for a cleaner design of the backend.

Alternative pointcut bindings AspectJ allows pointcuts to *bind* certain values from the context of a join point they match, and pass those values as parameters to advice bodies. For example, the pointcut *this(x)* binds *x* to the current value of **this** (and fails to match if the current method is static). Similarly, *target(x)* binds *x* to the receiver of a join point where this makes sense (*e.g.* in the case of a call to a non-static method). The variable *x* is declared to have a certain type, and if the appropriate runtime value is not of that type (or a subclass), the pointcut also fails to match.

One ambiguity in the existing specification of this feature is the treatment of pointcuts which combine variable binding and disjunction. Pointcuts such as $this(x) \parallel target(x)$ will bind *x* using the left disjunct if possible, and will try the right disjunct if that fails. We might now want to modify this pointcut to also impose an extra check on *x*, such as checking that as well as having its declared type it also implements some interface:

$(this(x) \parallel target(x))$ && **if**$(x$ **instanceof** $Serializable)$

The natural interpretation of such a pointcut would be to use backtracking to allow the second disjunct to be tried if the first one succeeds but the value of *x* is later rejected. We have defined such a backtracking semantics for these pointcuts, implementing it by rewriting all pointcuts to disjunctive normal form to avoid the runtime complexity of real backtracking. Currently, *ajc* forbids multiple pointcuts within the same expression from binding the same variable, but it is the intention of the *ajc* maintainers to implement the semantics we have defined in the future.

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 the control flow of the program reaches that shadow. This can be defined in terms of partial evaluation [13]. 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 Fig. 8(b), p. 15. 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 *this*, *target* and *args* can be given a variable name as an argument (such a variable must be declared with its type in the pointcut) or a type. In both cases, a runtime type check is inserted if it cannot be statically determined that the types match. If the variable is name, then the relevant value is exposed to the advice body.

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. At this stage, the join point shadows have been identified, and lists of possible advice applications have been computed for each shadow by the matcher. The weaver must insert code for advice and dynamic residues, ensuring that advice is run in the correct order at each join point.

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 earlier, *before* advice goes immediately after the beginning *nop* of the shadow (an example of this can be seen in Figure 8(b), p. 15), and all forms of *after* advice go immediately before the ending one. There are three forms of *after* advice: *after returning* advice runs on normal termination and is dual to *before* advice, while *after throwing* is implemented as an exception handler. Finally full *after* advice, which runs in both cases, is implemented by weaving both *after returning* and *after throwing*.

A novel strategy described in [11] 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.

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 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 Comparison with *ajc*

ajc is the original compiler for the AspectJ language, and it was written by the language's designers. It builds on a modified version of the Eclipse Java compiler, while the backend makes use of a customised version of BCEL. The design goals of *ajc* are quite different from those of *abc*: it aims to be a production compiler, with very short compile times and full integration with the Eclipse IDE. More information about *ajc*, including a detailed description of its weaver, can be found in [8]. By contrast, *abc*'s overriding design goals are extensibility and optimisation, as well as a complete separation from the components it builds on. In this section, we make a detailed comparison between the architecture of *ajc* and *abc*, in particular examining where the different design goals led to different design decisions.

6.1 Separation from components

To examine the way *ajc* and *abc* use their respective building blocks, we first measured their size in lines of code, making a distinction between the frontend and backend. The overall size of *ajc* and *abc* are comparable, as shown in the following table. These numbers were obtained in consultation with the authors of *ajc*, using the SLOCcount tool:

	ajc	*abc*
frontend	10,197	16,444
backend	23,938	17,397
total	34,135	33,841

At first glance it appears that *ajc*'s frontend is much smaller than that of *abc*. As we shall see shortly, this is achieved at the cost of making numerous changes in the source of Java compiler it builds on — and these changes are not listed here. Furthermore, *abc* uses Polyglot, which encourages the use of many tiny classes and requires a fair amount of boilerplate for visitors and factories. Another notable point in the above table is the small size of the backend of *abc*, which performs the most complex part of the compilation process (weaving). This is explained by the use of a clean intermediate representation, Jimple (which we present in more detail below in Section 6.3), as well as the rich set of analyses available in the Soot framework. We now examine in some detail how well *ajc* and *abc* are separated from the components that they build on.

Separation from base compiler: *ajc*. *ajc* builds on the Eclipse Java compiler. This compiler has been written for speed: for example, it eschews the use of Java's collection classes completely, in favour of lower-level data structures. It also uses dispatch on integer constants in favour of inheritance whenever appropriate.

Unfortunately, the architecture of the Eclipse compiler implies that *ajc* needs its own copy of the source tree of that compiler, to which local changes have been applied. These changes are by no means trivial: 44 Java files are changed, and there are at least 119 source locations where explicit changes are made. Furthermore, the grammar from which the Eclipse parser is generated has been modified. For pointcuts, the new parser simply reads in a string of "pseudo-tokens" that are then parsed by hand (using a top-down parser) in the relevant semantic actions.

The 119 changes that are made to the Java source are by no means trivial. For example, the class that implements Java's scope rules needs to be changed in 8 places. It is because of such changes to the Eclipse source tree that it can be fairly painful to merge *ajc* with the latest version of the Eclipse compiler.

Separation from base compiler: *abc*. By contrast, *abc* does not require any changes to the source of its base compiler, which is Polyglot. Polyglot has been carefully engineered to be extensible, and indeed *abc* is just another Polyglot extension. The changes to the scope rules are handled by introducing a new type for environments and a new type system. These are implemented as simple extensions of the corresponding classes in Polyglot. It is thus very easy to upgrade to new versions of Polyglot, even when substantial changes are made to the base compiler.

There are 14 types of AST node in Polyglot where it is necessary to override some small part of the behaviour. This is necessary, for example, because **this** has a different semantics in AspectJ when it occurs inside an intertype declaration. However, since Polyglot has been designed to allow changes of this nature to be made by subclassing, rather than by changing the source of Polyglot itself, no extra work is required when updating to a new version of Polyglot.

Finally, as we have described earlier, *abc* provides a clean LALR(1) grammar, presented in a modular fashion thanks to Polyglot's parser generator, which allows a neat separation between the Java grammar and that of an extension such as AspectJ.

Separation from bytecode manipulation: *ajc*. *ajc* uses BCEL, a library for directly manipulating bytecode, in order to perform weaving and code generation. As in the case of the base compiler, however, a special version of this library is maintained as part of the *ajc* source tree. Originally this was regularly synchronised with the BCEL distribution, using a patch file of about 300 lines. The specialised version is now developed as part of *ajc*, as BCEL is no longer actively maintained. The modified BCEL consists of 23,259 lines of code.

Separation from bytecode manipulation: *abc*. *abc* is completely separate from the Soot transformation and code generation framework; no changes to Soot are required whatsoever.

We conclude that *abc* is the first AspectJ compiler to achieve a clean-cut separation between the components it builds on. It seems likely that it will be possible to port the ideas that helped achieve this to extending other programming languages with aspect-oriented features.

6.2 Compile time

It is natural to enquire what the impact of using aspects is on the time taken to compile a program: an AspectJ compiler does a lot more work than a pure Java compiler. To assess this issue, we decided to compare four different compilers: normal *ajc*, *ajc* plus an optimisation pass of Soot over its output (*ajc + soot*), *abc* with all optimisations turned off (*abc -O0*), and *abc* with its default intraprocedural optimisations (*abc*). We measured compile times for seven benchmarks from [7], as shown in Table I. Our experiments were done on a dual 3.2GHz Xeon with 4BB RAM running Linux with a 2.6.8 kernel. We compiled using abc 1.0.1, soot 2.2.0, ajc 1.2.1 and javac 1.4.2.

benchmark	sloc	*ajc*	*ajc + soot*	*abc -O0*	*abc*	*javac*
bean	126	3.59	6.03	5.27	5.37	
bean-java	109	2.83	4.87	4.53	4.63	1.93
figure	94	3.39	5.33	4.49	4.76	
figure-java	98	2.82	4.58	3.90	4.36	2.03
nullcheck	1487	4.92	15.40	15.56	17.25	
nullcheck-java	1551	3.33	9.10	11.49	12.19	2.38
productlines	715	4.59	9.50	9.10	10.15	
authorization	685	3.24	7.37	8.02	9.35	
DCM	1668	5.49	22.09	20.88	24.60	
LoD	1586	7.10	58.18	29.46	42.83	

Table I: Compile times using *ajc*, *abc* and *javac* (seconds)

The first three AspectJ benchmarks (bean,figure,nullcheck) have Java equivalents, where the weaving has been performed by hand (bean-java,figure-java,nullcheck-java). As expected, aspect weaving has a significant impact on compile times. The main reason is that an AspectJ compiler needs to make a pass over all generated code to identify shadows and possibly weave in advice. It may be possible to curtail such a pass, for example by determining from information in the constant pool that no pointcut can match inside a given class. We plan to investigate such ways of reducing the extra cost of aspect weaving in future work.

The last four benchmarks make heavy use of aspects so there are no hand-woven Java equivalents. The productlines benchmark makes heavy use of inter-type declarations, while the others use mostly advice. Overall, the compile times indicate that *abc* is significantly slower than *ajc*. This is no surprise, as *abc*'s code has not been tuned in any way for compile time performance, whereas short compile times are an explicit design goal for *ajc*. The nullcheck benchmark is typical: the difference between *abc* and *ajc* for programs of a few thousand lines is usually a factor between 3 and 4. For examples where *abc* does a lot of optimisation, such as LoD, the gap can be a factor of 6. For very large inputs, such as *abc* compiling itself, the difference can be a factor of 14.

The compile times of *abc* reflect the cost of its powerful optimisation framework. In particular, an appropriate comparison is not with *ajc* (which lacks such optimisation capabilities), but with *ajc* + soot. This comparison shows that the compile times of *abc* and *ajc* + soot are very similar, which is encouraging.

It is furthermore pleasing that a research compiler such as *abc* can cope with very sizeable examples (such as compiling itself); we believe that one natural use of *abc* would be for optimised builds of programs whose day-to-day development is carried out with *ajc*.

6.3 Weaving into Jimple (*abc*) versus weaving into bytecode (*ajc*)

We illustrate the advantage of weaving into the three-address Jimple representation (as *abc* does) compared to weaving directly into bytecode (as *ajc* does) with a simple example of weaving a piece of advice before the call to method `bar` in the Java code shown in Figure 9(a). The results of weaving into this code both directly on bytecode and through Jimple are shown in Figure 9(b)-(d). In all cases, the instructions inserted in weaving are shown in boldface.

```
public int f(int x,int y,int z)
{
    return bar(x, y, z);
}
```

(a) base Java code

```
public int f(int x,int y,int z)
0:  aload_0
1:  iload_1
2:  iload_2
3:  iload_3
4:  istore %4
6:  istore %5
8:  istore %6
10: astore %7
12: invokestatic
     A.aspectOf ()LA;
15: aload  %7
17: invokevirtual
     A.ajc$before$A$124 (LFoo;)V
20: aload %7
22: iload %6
24: iload %5
26: iload %4
28: invokevirtual Foo.bar (III)I
31: ireturn
```

(b) direct weaving into bytecode (*ajc*)

```
public int f(int,int,int)
{ Foo this;
   int x, y, z, $i0;
   A theAspect;

   this := @this;
   x := @parameter0;
   y := @parameter1;
   z := @parameter2;
   theAspect = A.aspectOf();
   theAspect.before$0(this);
   $i0 = this.bar(x, y, z);
   return $i0;
}
```

(c) weaving into Jimple (*abc*)

```
public int f(int x,int y,int z)
0:  invokestatic A.aspectOf ()LA;
3:  aload_0
4:  invokevirtual
     A.before$0 (LFoo;)V
7:  aload_0
8:  iload_1
9:  iload_2
10: iload_3
11: invokevirtual Foo.bar (III)I
14: ireturn
```

(d) bytecode generated from Jimple (*abc*)

Figure 9: Weaving into bytecode versus weaving into Jimple

Figure 9(b) shows the bytecode for the method after the call to the before advice has been woven by *ajc*. Note that of the inserted bytecodes, only those as offsets 12 through 17 implement the lookup of the appropriate aspect and the call to the advice body. All of the remaining bytecodes are stack fix-up code that must be generated to fix up the implicit bytecode computation stack.

Figure 9(c) shows the Jimple code for the same method after the call to the before advice has been woven by *abc*. The key difference is that Jimple does not use an implicit computation stack. Instead, all values are denoted using explicit variables. Prior to weaving, the Jimple code is as in Figure 9(c), but without the three lines in boldface. To weave, *abc* needs only declare a Jimple variable, then insert the two lines to lookup the aspect and call the before advice. No additional code to fix up any implicit stack is needed.

Figure 9(d) shows the bytecode that Soot generates from the Jimple code from Figure 9(c). This bytecode has the same effect as the *ajc*-generated code in Figure 9(b), but it is significantly smaller because of Soot's standard backend optimisations. In addition, it uses only three local variables, compared to seven required by the *ajc*-generated code. We have observed that, even with modern JITs which perform register allocation, the excessive number of local variables required when weaving directly into bytecode has a significant negative impact on the performance of the woven code.

6.4 Using Soot Optimisations in Weaving

The use of Soot as a backend for *abc* enables it to leverage Soot's existing optimisation passes to improve the generated code. This simplifies the design of the weaver (see Section 5.4.2), but also enables aspect-specific optimisations that would be difficult or impossible to apply directly during weaving. In these cases, the Java optimisations are typically

augmented with AspectJ-specific information.

For example, AspectJ makes a special variable named *thisJoinPoint* available in advice bodies. This variable contains various reflective information about the join point that must be gathered at runtime and is relatively expensive to construct, so both *abc* and *ajc* implement "lazy" initialisation for this variable, so that it is only constructed when it will really be needed by an advice body, but that it is never constructed more than once even if more than one piece of advice applies at a join point. This is done by first setting the variable to *null*, then initialising it with the proper value just before advice is called, but only if it still contains *null*.

In *ajc*, the implementation does not work if there is any around advice at the join point (for technical reasons), and it is special-cased to avoid the unnecessary laziness if there is only one piece of advice at the join point. In *abc*, the lazy initialisation is used in all cases, and a subsequent nullness analysis is used to eliminate the overhead of the laziness in most cases (including the one where there is only one piece of advice). The analysis is a standard Java one, which has been given the extra information that the AspectJ runtime library method which constructs the *thisJoinPoint* object can never return *null*. Thus, the implementation is simpler and more robust than the *ajc* version.

6.5 Performance of object code

It is beyond the scope of the present paper to do a detailed comparison of the efficiency of code generated by *ajc* and *abc*. Because optimisations are an explicit design goal of *abc*, it is important that such experiments are thorough and realistic. In a companion paper [4], we provide a detailed account of the most important optimisations in *abc*, and of their effect on run times. The first of these is an improved implementation of **around** advice, giving a 6-fold speedup of on some benchmarks. The second is a set of intraprocedural improvements to **cflow**. Compared to version 1.2 of *ajc*, these yield improvements of 23 $\times$; and some of these optimisations have now been incorporated into *ajc* 1.2.1. Finally, we have implemented an interprocedural analysis to completely eliminate the cost of **cflow**, and this can lead to improvements of up to a 100-fold over *ajc* 1.2.1.

7 Related work

In the previous section we have provided a detailed comparison between the *ajc* AspectJ compiler and *abc*. The general strategy of weaving dynamic features in AspectJ, leaving dynamic residues where needed, is nicely explained in terms of partial evaluation in [13]. 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 strategies.

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* [9]. 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 [5], Jonas Bonér claims the overheads are negligible. To assess that claim, we translated a few benchmarks from [7] 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#) [18] and JAC (a framework for distributed aspect-oriented programming) [16]. 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 [6] 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.

8 Conclusions and Future Work

We have presented the design and implementation of the *abc* AspectJ compiler, building upon two existing compiler toolkits, Polyglot and Soot. *abc* 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.

Our principal contribution is to show how the architecture of *abc* was built around the Polyglot and Soot building blocks. This demonstrates how the AspectJ-specific information can be cleanly separated from the Java part (using the *AspectInfo* structure), enabling the use of Polyglot and Soot as Java tools. The *abc* compiler is the first AspectJ compiler to build on existing compiler tools without modification.

Building upon such powerful tools has had many benefits. The use of Polyglot as a frontend allowed a clean specification of the AspectJ grammar as an extension of the Java grammar, while providing mechanisms to implement the complex semantic checks required for AspectJ. Soot's Jimple intermediate representation allowed the *abc* weaver to be simpler and produce more efficient object code. Soot also provides a number of built-in optimisations to clean up woven code, and provides the opportunity to implement AspectJ-specific optimisations in future. The price for the use of these tools is a compile-time performance penalty, and we plan to investigate ways of reducing this in future work.

In seeking a clean implementation of AspectJ as an extension of Java, we have also clarified the AspectJ semantics, which has had a beneficial effect not just on *abc* but also on the *ajc* compiler.

Another 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. We believe that these implementation issues are relevant not just to AspectJ, but to compilers for other aspect-oriented languages, by highlighting the issues that arise in the implementation of such languages. In particular, the problems of ordering semantic checking phases in the presence of hierarchy introductions, determining and implementing scope rules for intertype declarations and designing an appropriate intermediate representation for advice matching are all important in the development of new aspect-oriented compilers.

Finally, *abc* incorporates a novel strategy for enabling painless optimisation of aspect-oriented constructs. This is achieved by allowing the weaving process to be repeated, so that naively woven code can be analysed to permit more precise weaving in subsequent passes. This is crucial tackling some of the performance issues inherent in aspect-oriented programming.

The abc group found the project of building the compiler to be exceptionally fun, challenging and educational. We have found that the *abc* architecture does meet our original goals of extensibility and optimisation – we have recently shown how to use *abc* to implement several language extensions [3] and we have also developed and implemented several optimisations [4]. Several other research groups are already using *abc* and we hope that *abc* will continue to be a research platform for further work on extending and compiling aspect-oriented languages. Our group is actively pursuing more 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` (to appear in AOSD 2005), 2004.

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

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

[6] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *GPCE '03*, volume 2830 of *LNCS*, pages 364–376, 2003.

[7] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *OOPSLA '03*, pages 149–168. ACM Press, 2003.

[8] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In K. Lieberherr, editor, *AOSD '04*. ACM Press, 2004.

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

[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP '01*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

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

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

[13] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In *CC '03*, volume 2622 of *LNCS*, pages 46–60, 2003.

[14] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *CC '03*, volume 2622 of *LNCS*, pages 138–152, 2003.

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

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

[17] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing Java using attributes. In *CC '01*, volume 2027 of *LNCS*, pages 334–554, 2001.

[18] 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.

[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 *CC '00*, pages 18–34, 2000.