abc : An extensible AspectJ compiler

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

Abstract. Research in the design of aspect-oriented programming languages requires a workbench that facilitates easy experimentation with new language features and implementation techniques. In particular, new features for AspectJ have been proposed that require extensions in many dimensions: syntax, type checking and code generation, as well as data flow and control flow analyses.

The AspectBench Compiler (*abc*) is an implementation of such a workbench. The base version of *abc* implements the full AspectJ language. Its frontend is built, using the Polyglot framework, as a modular extension of the Java language. The use of Polyglot gives flexibility of syntax and type checking. The backend is built using the Soot framework, to give modular code generation and analyses.

In this paper, we outline the design of *abc*, focusing mostly on how the design supports extensibility. We then provide a general overview of how to use *abc* to implement an extension. We illustrate the extension mechanisms of *abc* through a number of small, but non-trivial, examples.

We then proceed to contrast the design goals of *abc* with those of the original AspectJ compiler, and how these different goals have led to different design decisions. Finally, we review a few examples of projects by others that extend *abc* in interesting ways.

1 Introduction and Motivation

The design and implementation of aspect-oriented programming languages is a buoyant field, with many new language features being developed. In the first instance, such features can be prototyped in a system like the Aspect Sand Box [17] via a definitional interpreter. Such interpreters are useful in defining the semantics and in explaining the compilation strategy of new language features [32]. The acid test for new language features is, however, their integration into a full, industrial-strength language like AspectJ. That requires a highly flexible implementation of AspectJ that can be extended in a clean and modular way.

The purpose of this paper is to present *abc*, the AspectBench Compiler for AspectJ, which supports the whole of the AspectJ language implemented by *ajc* 1.2, and which has been specifically designed to be an extensible framework for implementing AspectJ extensions. *abc* is freely available under the GNU LGPL [1].

Challenges An AspectJ compiler is already a complex piece of software, which, in addition to the normal frontend and backend components of a compiler, must also support a matcher (for name patterns) and a weaver (both for intertype declarations and for advice). Furthermore, the kinds of extensions that have been suggested for AspectJ vary from fairly simple pointcut language extensions to more complex concepts which require modifications in the type system, matcher and weaver. To make the challenges explicit, we briefly review some previous work by others that has motivated our design.

At one end of the spectrum, there are fairly small extensions that require changes primarily to the syntax. An example of this kind is the *name pattern scopes* proposed by Colyer and Clement [12], which provide an abstraction mechanism for name patterns. To support this type of extension, our workbench needs an easy way of extending the syntax, as well as introducing named patterns into the environment.

A more involved extension is the *parametric introductions* of Hanenberg and Unland [23]. These are intertype declarations that depend on parameters evaluated at weave-time. Their integration into AspectJ requires substantial changes to the type system as well as the intertype weaver. This kind of extension thus motivates a highly flexible implementation of types.

Most proposals for new features in AspectJ are, however, concerned with the dynamic join point model. Sakurai *et al.* [39] propose *association aspects*. These provide a generalisation of per-object instantiation, where aspect instances are tied to a group of objects to express behavioural relationships more directly. This requires not only changes to the frontend, but also substantial changes to code generation. Making such code generation painless is another design goal of our workbench.

The community as a whole is concerned with finding ways of singling out join points based on semantic properties rather than naming. For instance, Kiczales has proposed a new type of pointcut, called *predicted cflow* [29]. **pcflow**(p) matches at a join point if there may exist a path to another join point where p matches. It is correct to let **pcflow**(p) match everywhere, but that would lead to inefficient programs. An efficient implementation of **pcflow**(p) needs substantial, interprocedural program analysis. Our workbench needs to provide a framework for building such analyses.

In fact, examples where efficient implementation necessitates an analysis framework abound. Particular instances include the *data flow pointcuts* of Masuhara and Kawauchi [31], and the *trace-based aspects* of Douence *et al.* [14], as well as the *communication history aspects* of Walker and Viggers [45].

All of the above are additions to the AspectJ language, but, of course, restrictions can be equally important in language design. One promising example is the proposal of Aldrich to restrict the visibility of join points to those that are explicit in the interface of a class [2]. We aim to support the implementation of such restrictions, and this requires a flexible implementation of the type system and the pointcut matcher.

Finally, we note that the implementation of advanced static checking tools for aspect-oriented programs, such as those investigated by Krishnamurthi *et al.* [30], require all types of extensions discussed above, ranging from simple variations in syntax to making advanced analyses such as escape analysis take into account the effects of advice.

In summary, we can see that an extensible AspectJ compiler must be able to handle a wide variety of extensions, possibly touching on many components of the compiler, including the frontend scanner and parser, the type checker, the matcher and weaver, and potentially requiring relatively sophisticated program analysis to ensure correctness and efficiency.

Design Goals One approach to implementing a language extension is to modify an existing compiler. However, this is not always the best approach, since existing compilers may not have been designed with extensiblity as one of the main goals. Furthermore, they may be constrained to work with infrastructures which themselves are not easily extensible. In the case of AspectJ, the only pre-existing implementation is *ajc*, which is designed to support fast and incremental compilation and also to interact closely with the Eclipse toolset.

Our approach was to design and implement *abc*, the AspectBench Compiler, with extensibility as its primary design goal. We also aimed for an optimising implementation of AspectJ, and we briefly summarise that perspective in our comparison with *ajc* in Section 6.5. To support extensibility, we distilled the following requirements from the above discussion of the challenges involved.

- simplicity: It must be relatively simple to develop new extensions. Users of the framework should not need to understand complicated new concepts or a complex software design in order to implement their extensions.
- modularity: We require two kinds of modularity. First, the compiler workbench itself should be very modular, so that the different facets of each extension can be easily identified with the correct module of the workbench.
  Second, the extension should be modular (separate from the workbench code). Users of the workbench should not need to change existing code; rather, they should be able to describe the extensions as specifications or code that is separate from the main code base.
- proportionality: Small extensions should require a small amount of work and code. There should not be a large overhead required to specify an extension.
- analysis capability: The compiler workbench infrastructure should provide both an intermediate representation and a program analysis framework. This is necessary for two reasons. First, some extensions may require relatively sophisticated analyses to correctly implement their semantic checks and weaving. Second, some extensions may lead to a lot of runtime overhead unless compiler optimisation techniques are used to minimise that overhead.

The abc approach To meet these objectives, we decided to build on existing, proven tools, namely the Polyglot extensible compiler framework for the frontend [37], and the Soot analysis and transformation framework for the backend [43]. [The McGill authors of the present paper are the authors of Soot.] Indeed, Polyglot has been shown to meet the criteria of simplicity, modularity and proportionality on a wide variety of extensions to the syntax and type system of Java. By the same token, Soot has been shown to meet all the above criteria for code generation, analysis and optimisation.

Given the success of these building blocks, we felt it extremely important to design *abc* so that both are used *as is*, without any changes that are specific to *abc*, in order to allow easy migration to new releases of those frameworks. As explained in Section 2 below, this has dictated an architecture where the frontend separates the AspectJ program into a pure Java part and a part containing instructions for the backend.

Contributions In general terms, the contributions of this paper are the following:

- comprehensive account of an AspectJ compiler: While *ajc* has been in use for eight years or more, there are few publications that give a comprehensive account of its main design decisions, a notable exception being the description of its advice weaver in [27]. The present paper aims to provide a general overview of how to build an AspectJ compiler, while pointing out the structure that is common to *ajc* and *abc*. We also examine the consequences of the different design goals of *ajc* and *abc*, in particular how *abc* places more emphasis on extensibility and optimisation.
- extensible workbench for AOP research: We have identified the requirements for a workbench for research in aspect-oriented programming languages by analysing previous research in this area. We show how *abc* meets these requirements, and validate our architecture with a number of small but non-trivial examples. Furthermore, we present an overview of extensions to *abc* that have been implemented by other researchers.
- experience with Soot and Polyglot: *abc* builds on Polyglot and Soot without making any changes to these two components. As such, *abc* is one of the largest projects undertaken with either Soot or Polyglot. This paper is therefore also an experience report, assessing the suitability of Polyglot and Soot for building aspect-oriented programming tools.

At a more technical level, the contributions of *abc* with respect to extensibility are these:

- pass structure: *abc* has a carefully designed pass structure, where each compiler pass achieves exactly one task, so that it is never necessary to split an existing pass when inserting a new one required by an extension. Designing such a pass structure that processes all types in the right order is quite hard, as witnessed, for example, by a bug concerning ITDs on inner classes in *ajc* [5]. Another example is the need for three separate passes that evaluate classname patterns. The pass structure is outlined in Section 2, and then further detailed as necessary for our examples.
- separator: A *separator* pass that splits the original AspectJ AST into a pure Java part and the aspectinfo; by enforcing that separation very strictly, extensions never need to modify the code generation pass, which is used unchanged from the Soot framework. The separator is explained in Section 2.3.
- use of Jimple: The use of a typed, stackless, 3-address intermediate representation, namely Jimple, to significantly simplify doing a good job of writing a new weaver for new joinpoint types. The advantages of Jimple (versus bytecode) for weaving are discussed in Section 6.3.
- regular IR for pointcuts: An intermediate representation of pointcuts that is more regular than at source level. This representation makes it easier to represent new pointcut primitives, and we shall illustrate this with the example of local pointcut variables. The intermediate representation includes reducing complex pointcut expressions to disjunctive normal form. An added benefit is that it sorted out some

nettly problems with the treatment of disjunction (||) in *ajc* [4]. Our intermediate representation for pointcuts is presented in Section 3.6.

reweaving: An explicit representation of residues via a meta-language that can be optimised based on further analysis of woven Jimple; and a re-entrant design of the weaver to exploit such opportunities via a weave-analyse-weave cycle. This reweaving architecture enables easy plug-and-play of complex optimisations. This architecture is first introduced in Section 2.4, and we present some numbers that demonstrate its advantages in Section 6.5.

Paper Structure The structure of this paper is as follows. In Section 2, we first give an overview of the main building blocks of *abc*, namely Polyglot and Soot, and show their role in the overall architecture of *abc*. Next, in Section 3 we sketch the main points of extensibility in *abc*. We then turn to describe some modest but representative examples of AspectJ extensions in Section 4, and their implementation in Section 5. The design goals of *abc* are contrasted with those of the original AspectJ compiler *ajc* in Section 6, and we examine how the different goals have led to different design decisions. A particular topic highlighted in Section 6 is the use of Jimple in a weaver, why it is good for extensions and for implementing optimisations. In Section 7, we review a few examples by other researchers who have extended *abc*. Also in Section 7, we discuss a number of similar projects that share *abc*'s goals. Finally, in Section 8 we draw some conclusions from our experience in building *abc*, and we explore possible directions for future research.

This paper is an enhanced, updated version of [6]. New material includes: the architecture of the weaver in Section 2, a detailed qualitative comparison to *ajc* in Section 6, a discussion of other projects that build on *abc* in Section 7, and many small improvements throughout.

2 Architecture

As stated in the introduction, *abc* is based on the Polyglot extensible compiler framework [37] and the Soot bytecode analysis and transformation framework [43]. Using Polyglot as an extensible frontend enables customisation of the grammar and semantic analysis; in the backend, Soot provides a convenient intermediate representation on which to implement the weaving of extensions, as well as tools for writing any program analyses that extensions may require.

Input classes can be given to *abc* as source code or class files, and *abc* is able to weave into both. Source files are processed by the Polyglot frontend, whereas only the signature part of class files are read by Polyglot in order to perform type checking of the source code. In both cases, weaving is performed on Jimple, Soot's intermediate representation.

Because *abc* works with an unmodified Soot and Polyglot, it is easy for us, as the developers of *abc* itself, to update to the latest versions of Soot and Polyglot as they are released. By the same token, authors of AspectJ extensions can upgrade to new versions of *abc* without difficulty. This independence was achieved mainly by separating the AspectJ-specific features in the code being processed from standard Java code. In

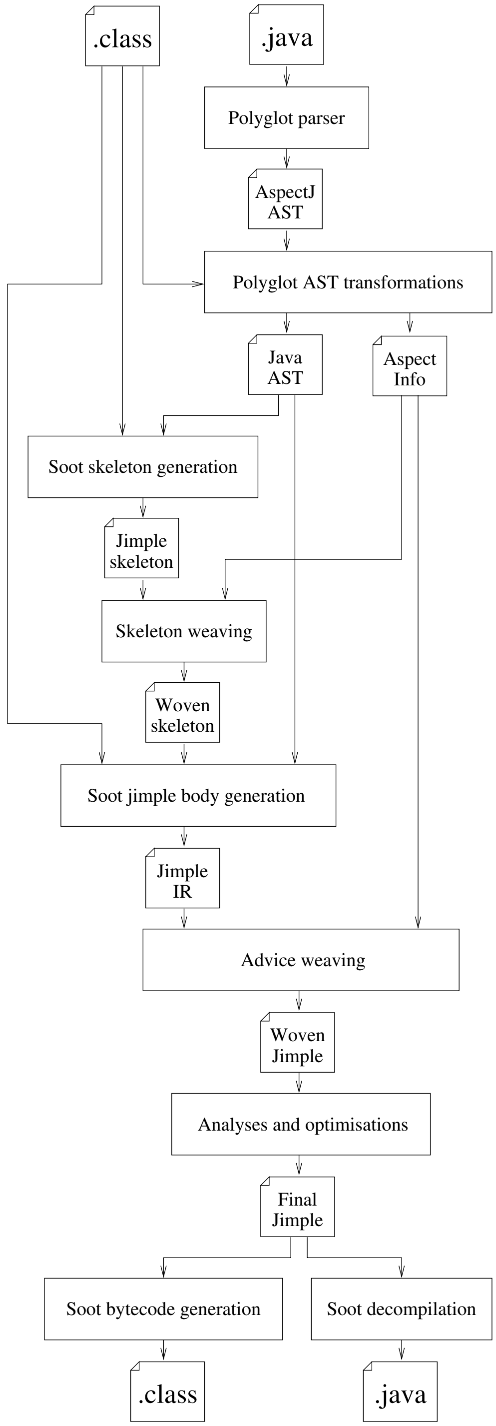

Fig. 1. *abc* overall design

the frontend, *abc* generates a plain Java abstract syntax tree (AST) and a separate aspect information structure containing the aspect-specific information. We call the aspect information structure the *AspectInfo*. The unmodified backend can read in the AST (because it is plain Java), and *abc* then uses the *AspectInfo* to perform all required weaving. A simplified diagram of the architecture of *abc* is shown in Figure 1. In many respects, this architecture is similar to that of *ajc*. At this level of abstraction, the main difference is the strict use of a separator pass (labelled 'Polyglot AST transformations' in the figure) for splitting the pure Java from any aspect-specific information. This separation process is described in more detail below.

In the following subsections, we describe Polyglot and Soot in the context of *abc*, with a focus on how they contribute to extensibility. Finally we discuss in some more detail how the two parts are connected.

2.1 Polyglot

Polyglot [37] is a frontend for Java intended for implementing extensions to the base language. In its original configuration, Polyglot first parses Java source code into an abstract syntax tree (AST), then performs all the static checks required by the Java language in a number of passes which rewrite the tree. The output of Polyglot is a Java AST annotated with type information, which is written back to a Java source file. Polyglot is intended to perform all compile-time checks; when a class has passed through all of the passes in Polyglot, the resulting Java file should be compilable without errors by any standard Java compiler. When Polyglot is used as a frontend for Soot, the *Java to Jimple* module inside Soot compiles the final AST into the Jimple intermediate representation instead of writing it out to a Java file. Therefore, in *abc*, the final Polyglot passes separate the AspectJ program into pure Java (which is passed to the Java to Jimple module in Soot) and instructions for the backend.

Several features of Polyglot make it well-suited for writing extensions, and also help to make those extensions themselves extensible. Polyglot allows a new grammar to be specified as a collection of modifications to an existing grammar, where these modifications are given in a separate specification file, not in the original grammar file. The AspectJ grammar we developed for *abc* is specified as an extension of the Java grammar, and the grammars for extensions are in turn specified as modifications to the AspectJ grammar.

Polyglot makes heavy use of interfaces and factories, making it easy to extend or replace most of its parts, such as the type system or the scope rules, as well as the list of rewrite passes that are performed on the AST. Each pass in Polyglot non-destructively rewrites the input tree. As a result, it is easy to insert new passes in between existing ones, and each pass typically performs only a small amount of work compared to traditional compiler passes. In *abc*, we have added many AspectJ-specific passes, and it is easy for extensions to add further passes of their own. The ordering of passes must be chosen carefully, since the semantic analysis of Java source code might depend on changes to the program introduced by aspects.

Each AST node in Polyglot uses a mechanism of *extensions* and *delegates* to allow methods to be replaced or added in the middle of the existing class hierarchy, achieving an effect similar to what can be done in AspectJ using intertype declarations, but in plain Java. This mechanism is commonly used by extensions of *abc* to modify existing AST nodes.

2.2 Soot

Soot [43], which is used as the back-end of *abc*, is a framework for analysing and transforming Java bytecode. The most important advantage of using Soot as the backend, both for developing *abc* itself and for extending the language, is Jimple, Soot's intermediate representation. Soot provides modules to convert between Jimple, Java bytecode, and Java source code. It furthermore includes implementations of standard compiler optimisations, which *abc* applies after weaving. We have already observed significant speedups from these optimisations alone [7]. In addition to already implemented analyses and transformations, Soot has tools for writing new ones, such as control flow graph builders, definition/use chains, a fixed-point flow analysis framework, and a method inliner. These features are useful for implementing extensions that need to be aware of the intra-procedural behaviour of the program, such as pointcuts describing specific points in the control flow graph.

The Jimple intermediate representation is a typed, stack-less, three-address code. Rather than representing computations with an implicit stack, each Jimple instruction explicitly manipulates specified local variables. This representation simplifies weaving of advice, both for standard AspectJ features and for extensions. If it were weaving into bytecode directly, the weaver would need to consider the effect of the woven code on the implicit execution stack, and generate additional code to fix up the stack contents. None of this is necessary when weaving into Jimple. Moreover, when values from the shadow point are needed as parameters to the advice, they are readily available in local variables; the weaver does not have to sift through the computation stack to find them.

As input, Soot can handle both class files and Java source files. To convert bytecode to Jimple, Soot introduces a local variable to explicitly represent each stack location, splits the variables to separate independent uses of the same location, and infers a type [20] for each variable. To convert source code to Jimple, Soot first uses Polyglot to construct an AST with type information, and then generates Jimple code from the AST. This process does not need to be modified in *abc*, because *abc* passes Soot a plain Java AST, keeping all the aspect-specific information in the separate aspect information structure. Normally, after all processing, Soot converts the Jimple code into bytecode and writes it to class files, but it also includes a decompiler, *dava* [33], which is very useful for viewing the effects of aspects and AspectJ extensions on the generated code.

2.3 Connecting Polyglot and Soot

We conclude the discussion of *abc*'s architecture by examining in closer detail how Polyglot and Soot interact. A key component of this interaction is the separation of the AspectJ AST into a pure Java AST and the auxilliary *AspectInfo* structure. This transformation enables *abc* to use the existing facility in Soot for translating a Polyglot AST into the Jimple IR. This is an important design decision in *abc*, as it implies that extension writers never need to modify the existing code generator. Other aspect-oriented systems that use a similar separation pass include AspectWerkz and Hyper/J [8, 38].

The Java AST is basically the AspectJ program with all AspectJ-specific language constructs removed. The *AspectInfo* structure contains complete information about these constructs. In cases where these contain actual Java code (advice bodies, **if** pointcut conditions, intertype method/constructor bodies, intertype field initialisers), the code is placed in placeholder methods in the Java AST.

The Java AST only contains Java constructs, but it is incomplete in the sense that it may refer to class members which do not exist or are not accessible in the unwoven Java program. More specifically, the Java AST will in general not be compilable until all *declare parents* and intertype declarations have been woven into the program. The first of these can alter the inheritance hierarchy, and the second can introduce new members that the pure Java parts may refer to. Since both of these features may be applied to class files (for which we do not have an AST representation), it is not possible to perform this part of the weaving process on the Polyglot representation before passing the AST to Soot.

Fortunately, Soot allows us to conduct the conversion from Java to Jimple in two stages, and the application of *declare parents* and intertype weaving can happen in between. In the first stage, Soot builds a class hierarchy with mere stubs for the methods: it is a skeleton of a full program in Jimple, without method bodies. In the second stage, Soot fills in method bodies, either by converting bytecode from class files, or by compiling AST nodes.

This setup permits both static weaving and advice weaving to work on the Jimple IR, largely independent of whether the Jimple code was generated from source code or bytecode. And since the skeleton that is filled out in the second stage has the updated hierarchy and contains all intertype declarations, all member references in the code are resolved correctly in the translation into Jimple.

The two-stage weaving (static and advice) is shared with *ajc*. Indeed, the two stages are dictated by the AspectJ language design: static weaving only effects the type hierarchy, whereas advice weaving effects runtime behaviour. Furthermore, one cannot generate code without first adjusting the type hierarchy.

2.4 The advice weaver

The job of the advice weaver is to modify the Jimple code according to the instructions in the *AspectInfo* such that advice bodies are executed whenever the corresponding pointcuts match the currently executing join point.

The architecture of the advice weaver is shown in Figure 2. The first step of advice weaving is to identify all *join point shadows*, that is, all places in the Jimple code that could potentially correspond to a join point in the execution of the program. Each of these are then matched against all pointcuts in the program. If it is determined that a pointcut might match a join point at a particular shadow, the matcher emits a weaving instruction telling the weaver to weave the advice body at that shadow. Since a pointcut can contain terms that depend on the runtime state of the program, it cannot always be fully determined at compile time whether a particular pointcut matches at a shadow. A weaving instruction thus consists of three parts: the shadow at which to weave, the advice to weave in and a *dynamic residue* specifying what additional runtime checks must be inserted to check that the pointcut actually matches the current join point. The

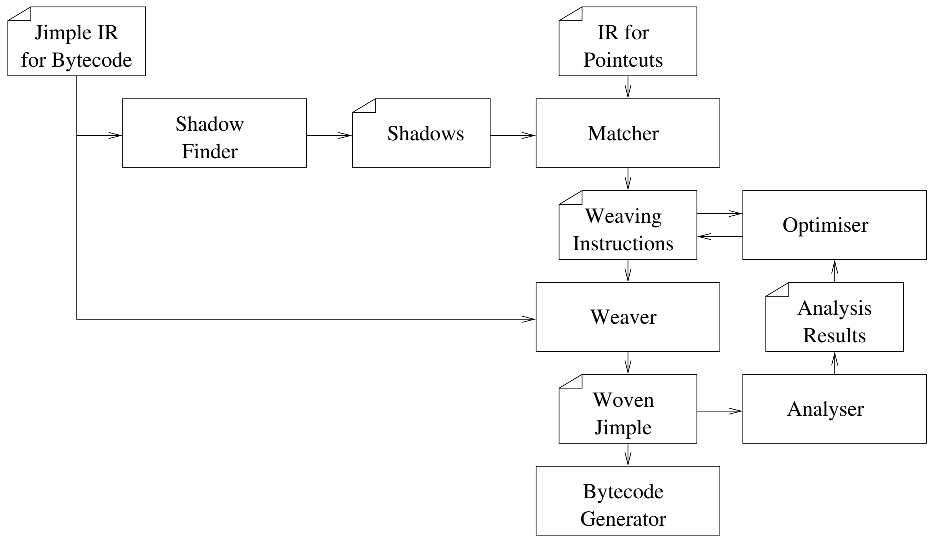

Fig. 2. Design of the *abc* advice weaver

dynamic residue also contains information about how to bind the values that are to become the arguments to the advice.

When all weaving instructions have been generated, the actual weaving is performed. The result is a Jimple program whose behaviour includes all advice bodies executing at the appropriate times. This program is then translated into bytecode by the Soot bytecode generator and the result is written out to the target class files.

Some extensions might require some sophisticated analysis to be done on the Jimple code. These fall roughly into two categories: *pre-weaving analysis*, where the analysis is performed on the original Java code before the advice are woven in, and *post-weaving analysis*, where the analysis is performed on the woven code. Pre-weaving analysis is typically employed when the analysis results are needed by the pointcut matcher, for instance when implementing a new kind of pointcut. Post-weaving analysis is used when some property of the final code is desired, for instance when doing optimisations on the final code or checking behavioural properties of the program.

In some cases, such as the **cflow** optimisation mentioned in Section 6.5, the analysis needs to be performed on the woven code, but the result is needed by the weaver. To facilitate such analyses, *abc* includes a mechanism for *re-weaving*, which can throw away the woven code and revert to the unwoven code while retaining the analysis results obtained from analysing the woven code. This is also illustrated in Figure 2. The results of the analysis are channelled back into an optimisation pass which modifies the weaving instructions to be used in a subsequent weaving pass. This process can be repeated as many times as necessary.

3 Defining an Extension

We now outline the basic steps needed to create an extension, in a general manner. This description is intended to give the reader an impression of the extension mechanisms available in *abc*, without delving into excessive detail. After this generic description, we shall introduce some concrete examples in Section 4, and show how the basic steps are instantiated in Section 5.

This section serves two purposes. First, to outline how we build on the existing extension mechanisms of Polyglot and Soot to achieve extensibility in *abc* (Sections 3.2, 3.3, 3.4, and 3.8). Second, we wish to present some design decisions that are unique to *abc*, which address specific issues regarding the extension of AspectJ (Sections 3.1, 3.5, 3.6, 3.7 and 3.9).

3.1 Syntax

The first step in implementing a new extension is usually defining what additional syntax it will introduce to the language. Making *abc* recognise the extended language involves changing the lexer and the parser that it uses. Polyglot already handles extending grammars in a very clean and modular fashion. However, the standard Polyglot lexer is not extensible — extensions are expected to create their own lexer by copying it and making appropriate modifications. Thus, in this subsection we describe our approach to making an extensible lexer in some detail, and then briefly summarise the Polyglot mechanism for extending grammars.

Lexer We have designed the lexer for *abc* to support a limited form of extensibility that has been sufficient for the extensions we have written so far. Specifically, the set of keywords recognised by the lexer can be modified by an extension, and the actions taken by the lexer when encountering one of these keywords are customisable. More complex extensions can still be achieved by reverting to Polyglot's approach of copying and modifying the lexer definition. This is in agreement with the principle of proportionality which was stated as a design goal — small extensions are easy, and complex ones are possible.

The lexical analysis of AspectJ is complicated by the fact that there are really several different languages being parsed: ordinary Java code, aspect definitions, and pointcut definitions. Consequently, the *abc* lexer is stateful — it recognises different tokens in different contexts. The following example illustrates one kind of problem that is dealt with by the introduction of lexer states:

*if**.*1.*Foo*+.*new*(..)

The expected interpretation of such a string as Java code and as part of a pointcut will be very different; for example, in Java, we would expect "1." to become a floating point literal, whereas in the pointcut language the decimal point would be viewed as a dot separating elements of a name pattern. Similarly, "*" in Java should be scanned as an operator, while in pointcuts, it is part of a name pattern. Note also the use of what would be keywords in Java mode (**if** and **new**) as part of a pattern.

An important part of designing a stateful lexer is specifying when the lexer should switch to a different state without adding too much complexity. The general pattern we use is to maintain a stack of states, and recognise the end of a state when we reach an appropriate closing bracket character for that state. For example, normal Java code is terminated by the '}' character. Of course, braces can be nested, so we need to recognise opening braces and also count the nesting level. For more details regarding the lexer states in *abc*, see section 5.2.

Parser The *abc* parser is generated by PPG [9], the LALR parser generator for extensible grammars which is included in Polyglot [37]. PPG allows changes to an existing grammar to be entered in a separate file, overriding, inheriting and extending productions from the base grammar. This results in modular extensions, which can easily be maintained should the base grammar change.

The example in Figure 3 (using simplified non-PPG syntax) demonstrates the basic principles. An existing grammar can be imported with the "*include*" keyword. New production rules can then be specified, and one can change existing rules using the keywords "*extend*" and "*drop*" to add and remove parts of the rule. More advanced changes, such as modifying the precedence of operators, are also possible. For further details on the specification of grammar, see [9].

File **X**	File **Y**
S ::= a | b | c	*include* **X** *extend* S ::= d | e
File **Z**	**Result**
include **Y** *drop* S ::= b | d	S ::= a | c | e

Fig. 3. Grammar extension mechanism

3.2 Type system

Polyglot provides convenient facilities for extending the type system. As a minimum, this involves introducing a new kind of type object and lookup functions for these new entitities in the environment. The new type of environment is then invoked by overriding the environment factory method in a subclass of *AJTypeSystem*, which describes the type system of AspectJ itself.

To illustrate, consider the introduction of named class pattern expressions [12]. We would need to introduce a new type object to represent such names, say *NamedCPEInstance* (in Polyglot, it is convention that identifiers for type classes end with ...*Instance*). The environment then maps (possibly qualified) names to objects of type *NamedCPEInstance*.

The semantic checks for named patterns must enforce the requirement that there be no cycles in definitions, since recursively defined named patterns do not make sense. A similar check has already been implemented for named pointcuts, and it involves building a dependency graph. Such data structures necessary for semantic checks are typically stored in the type objects (here *NamedCPEInstance*): because Polyglot operates by rewriting the original tree, it is not possible to store references to AST nodes.

Examples such as the parametric introductions of Hanenberg and Unland [23] would require more invasive changes in the type system, for example by subclassing *InterTypeMethodInstance* (the signature of a method introduced via an intertype declaration) to take account of the parameters that are to be evaluated at compile time.

3.3 Semantic checks

New semantic checks are usually implemented by overriding the appropriate method on the relevant AST nodes. The most obvious place for simple checks is in the *TypeChecker* pass; every AST node implements a *typeCheck*(*TypeChecker*) method. The type checker is run after all variable references are resolved; all checks that do not require further data structures are typically put in the *typeCheck* method.

Later passes use data flow information to check initialisation of local variables and the existence of **return** statements. Again, each AST node implements methods to build the control flow graph for these purposes. In the base AspectJ implementation, these are, for example, overridden to take into account the initialisation of the result parameter in **after returning** advice, and extensions can make variations of their own.

AspectJ is somewhat unusual in that some semantic checks have to be deferred to the weaver. For example, it is necessary to type check the results of **around** advice at each point where it is woven in. Because *abc* maintains precise position information throughout the compilation process, such errors can still be reported at the appropriate locations in the source.

3.4 Rewriting

The normal use of Polyglot is as a source-to-source compiler for extensions to Java, where the final rewriting passes transform new features into an equivalent pure Java AST. *abc* is different in that most of the transformation happens at a later stage, when weaving into Jimple. It is, however, often useful to employ Polyglot's original paradigm when implementing extensions to AspectJ that have an obvious counterpart in AspectJ itself.

For example, consider again the feature of named class pattern expressions. A simple implementation would be to just inline these after appropriate semantic checks have been done, so that nothing else needs to change in the compiler. Such inlining would be implemented as two separate AST rewriting passes, one to collect the named pattern definitions and the other to inline them — the two would then communicate via an explicit data structure that is common to both passes. As said, it is not recommended to store pieces of AST explicitly unless they are immediately transformed away.

abc does extensive rewriting of the AST prior to conversion to Jimple. This consists of introducing new placeholder methods (for instance for advice bodies), and storing instructions for the backend in the *AspectInfo*. Extensions can participate in this process by implementing methods that are called by the relevant passes.

3.5 Join points

Introducing new pointcuts will often involve extending the set of possible join points. For example, implementation of a pointcut that matches when a cast instruction is executed would require the addition of a join point at such instructions.

Many new join points will follow the pattern of most existing AspectJ join points and apply at a single Jimple statement. These can be added by defining a new factory class that can recognise the relevant statements, and registering it with the global list of join point types.

For more complicated join points, it will be necessary to override the code that iterates through an entire method body looking for join point shadows. The overriding code can do any required analysis of the method body to find instances of the new join points (for example, one might want to inspect all control flow edges to find the back edges of loops [25]), and then call the original code to find all the "normal" join point shadows.

3.6 Pointcuts

As pointed out in the introduction, there are many proposals for new forms of pointcuts in AspectJ. To meet our objective of proportionality (small extensions require little work), we have designed an intermediate representation of pointcuts that is more regular than the existing pointcut language of AspectJ. This makes it easier to compile new pointcut primitives to existing ones.

Specifically, the backend pointcut language partitions pointcuts into the four categories listed below. Some of the standard AspectJ pointcuts fit directly into one of these categories and are simply duplicated in the backend, while others must be transformed from AspectJ into the representation used in *abc*.

- Lexical pointcuts are restrictions on the lexical position of where a pointcut can match. For example **within** and **withincode** fall into this category.
- Shadow pointcuts pick out a specific join point shadow within a method body. The **set** pointcut is an example.
- Dynamic pointcuts match based on the type or value of some runtime value. Pointcuts such as **if**, **cflow** and **this** are of this kind.
- Compound pointcuts represent logical connectives such as &&.

The motivation for this categorisation is that it allows the implementation of each backend pointcut to be simpler and more understandable, which in particular makes it easier for extension authors to define new pointcuts.

An example of an AspectJ pointcut that does not fit into this model directly is the **execution**(⟨*MethodPattern*⟩) pointcut, which specifies both that we are inside a method or constructor matching *MethodPattern* and that we are at the execution join point. The backend pointcut language therefore views this as the conjunction of a lexical pointcut and a shadow pointcut.

To add a new pointcut, one or more classes should be added to the backend, and the frontend AST nodes should construct the appropriate backend objects during the generation of the *AspectInfo* structure.

The key class of the *AspectInfo* is the *GlobalAspectInfo* class — this is a singleton (it has precisely one instance during a compiler run), and it contains lists of advice declarations, intertype declarations, and so on. It also contains mappings to retrieve the precedence of two aspects, to find the non-mangled name of a private intertype declaration, and many similar mappings. The frontend inserts the appropriate information into these data structures via the accessor functions provided by *GlobalAspectInfo*. The *AspectInfo* also contains classes for the intermediate pointcuts, and the class hierarchy for these closely follows the above description.

The backend classes are responsible for deciding whether or not the pointcut matches at a specific location. If this cannot be statically determined, then the pointcut should produce a *dynamic residue* which specifies the generation of the required runtime code.

3.7 Advice

It appears that there are few proposals for truly novel types of advice: most new proposals can be easily rewritten to the existing idioms of before, after and around. For example, the proposal for "tracecuts" [14, 45] reduces to a normal aspect, where a state variable tracks the current matching state, and each pattern/advice pair translates into after advice. Such new types of advice are thus implemented via rewriting, in the standard paradigm of Polyglot.

Nonetheless, adding a new kind of advice that follows the AspectJ model of advice is straightforward: simply implement a new class and define how code should be generated to call that piece of advice and where in the join point shadow this code should go. For example, the bookkeeping required for **cflow** is implemented as a special kind of advice that weaves instructions both at the beginning and end of a shadow.

3.8 Optimisations

The straightforward implementation of a new extension may result in inefficient runtime code. Even in the basic AspectJ language, there are a number of features that incur significant runtime penalties by default, but in many cases can be optimised. *abc* aims to make it as easy as possible to implement new optimisations, whether for the base language or for extensions. In particular, it is straightforward to transform the AST in the frontend and the Jimple intermediate code in the backend.

Taking an example from the base AspectJ language, construction of the **thisJoinPoint** is expensive because it must be done each time a join point is encountered at runtime. *abc* (like *ajc*) employs two strategies for mitigating this overhead. Firstly, some advice bodies only ever make use of the *StaticPart* member of **thisJoinPoint**, which only needs to be constructed once. A Polyglot pass in the frontend is used to identify advice bodies where this is the case and transform the uses to **thisJoinPointStaticPart** instead.

Secondly, the runtime code generated delays construction until as late as possible in case it turns out not to be needed at all; this is complicated by the fact that *if* pointcuts as well as advice bodies may make use of it, so construction cannot simply be delayed until the advice body runs. *abc* generates code that instantiates the **thisJoinPoint** variable where needed *if it has not already been instantiated*, using *null* as a placeholder until that point. The Jimple code is then transformed to remove unnecessary checks and initialisations, using a variation of Soot's intraprocedural nullness analysis which has special knowledge that the **thisJoinPoint** factory method cannot return *null*.

3.9 Runtime library

The runtime library for AspectJ serves two purposes. Firstly, it contains bookkeeping classes necessary for the implementation of language constructs such as **cflow**. Extensions such as data flow pointcuts [31] would require a similar runtime class in order to store dynamic data about the source of the value in a particular variable.

Secondly, the runtime provides the objects accessible through the *thisJoinPoint* family of special variables; these make information about the current join point available to the programmer via reflection. Any new pointcut introduced is likely to have unique signature information which would be accessible to the user via an extension of the *Signature* interface. For example, the standard AspectJ runtime contains, amongst others, *AdviceSignature*, *FieldSignature*, and *MethodSignature*.

4 eaj — An AspectJ extension

This section describes a few particular extensions to the AspectJ language that we have implemented. These extensions have been chosen to illustrate the most salient of the mechanisms that were described in the previous section. The full source code for these examples is included with the standard distribution of *abc* [1]. For ease of reference, the extended language is named *eaj*; one compiles *eaj* programs with the command '*abc -ext abc.eaj*'. This is the usual way of invoking extensions with *abc*.

4.1 Private pointcut variables

In AspectJ, the only way to introduce new variables into a pointcut is to make them explicit parameters to a named pointcut definition or advice. It is sometimes convenient, however, to simply declare new variables whose scope is only part of a pointcut expression, without polluting the interface of the pointcut. For example, it might be desired to check that the value of an argument being passed has certain properties, without actually using that value in the advice body. The new keyword **private** introduces a locally scoped pointcut variable. For instance, the following pointcut could be used to check that the argument is either a negative *int* or a negative *double*:

```
pointcut negativefirstarg() :
        private (int x) (args(x) && if(x < 0))
     || private (double x) (args(x) && if(x < 0));
```

4.2 Global pointcuts

It is very common for many pieces of advice to share a common conjunct in their pointcut. The idea of a *global* pointcut is to write these common conjuncts only once. An example use is to restrict the applicability of every piece of advice within a certain set of aspects. For example, we might write:

global : * : !**within**(*Hidden*);

This would ensure that no advice within any aspect could apply within the *Hidden* class.

As another example, it is often useful to prevent advice from an aspect applying within that aspect itself. The following declaration (for aspect *Aspect*) can achieve this more concisely than putting the restriction on each piece of advice:

global : *Aspect* : !**within**(*Aspect*);

In general, a global pointcut declaration can be put anywhere a named pointcut declaration can be (*i.e.*, directly within a class or aspect body). The location of such a declaration has no effect on its applicability, except that name patterns within such a declaration will only match classes and aspects visible from the scope of that declaration.

The general form of a global pointcut declaration is as follows:

global : ⟨*ClassPattern*⟩ : ⟨*Pointcut*⟩ ;

It has the effect of replacing the pointcut of each advice declaration in each aspect whose name matches *ClassPattern* with the conjunction of the original pointcut and the global *Pointcut*.

4.3 Cast pointcuts

The purpose of the *cast* pointcut is to match whenever a value is cast to another type. A corresponding new type of join point shadow is added which occurs at every cast instruction, whether for reference or primitive types, in the bytecode of a program.

To illustrate, the following piece of advice can be used to detect runtime loss of precision caused by casts from an *int* to a *short*:

```
before(int i):
        cast(short) && args(i)
    && if(i < Short.MIN_VALUE
          || i > Short.MAX_VALUE)
{
    System.err.println("Warning: loss of " +
                       " precision casting " +
                       i + " to a short.");
}
```

In general the syntax of a **cast** pointcut is **cast**(⟨*TypePattern*⟩); this will match at any join point where the static result type of the cast is matched by *TypePattern*. In keeping with the pattern of other primitive pointcuts, the value being cast from can be matched by the **args** pointcut, and the result of the cast can be matched by the optional parameter to **after returning** advice (and is returned by the **proceed** call in **around** advice).

4.4 Throw pointcuts

The **throw** pointcut is introduced in the developer documentation for *ajc* [28], and we have implemented it in *eaj* to compare the ease-of-extension of both compilers. It matches a new join point shadow which occurs at each throw instruction.

The following example demonstrates how extended debugging information can be produced in the event of a runtime exception, using a piece of advice:

```
before(Debuggable d):
    this(d) && throw() && args(RuntimeException)
{
    d.dumpState();
}
```

5 Implementing eaj using abc

We have given a broad outline of how extensions are constructed and discussed some specific extensions that we have implemented. We now show in detail how this was done, both to provide a guide for others and to enable a realistic assessment of the work involved.

5.1 Roadmap

As we do not wish to hide any of the difficulties involved in writing an *abc* extension, the presentation in the next few subsections is necessarily somewhat technical, so let us first outline a generic roadmap of an *abc* extension. This will provide readers with a high-level structure for the detailed explanations that follow.

Extension packages An extension typically consists of five Java packages, plus two new 'driver' classes that bind the extension to the existing base compiler. The five relevant packages are shown in Figure 4. The first of these is concerned with syntax, and serves to introduce new keywords and grammar rules: these will be discussed in Sections 5.2 and 5.3 below. Next, one needs to write new classes for AST nodes. In Section 5.4 we give an overview of what this involves for the example *eaj* extension. It is quite common that new language features require new compiler passes. For the running example, that is the case with global pointcuts, as it is necessary to collect all of these to make appropriate modifications to advice declarations. In Section 5.5 we show how to write a new pass for this purpose. This also requires subclassing the existing AST representation of advice declaration: such subclasses reside in the *extension* package. For the simple examples in this paper, it is not necessary to extend the AspectJ type system. All extensions to the backend of the compiler occur in subpackages of *weaving*. Readers may wish to glance back at Figure 2 which depicts the architecture of *abc*'s weaver. For the example in hand, one needs to extend the intermediate representation for pointcuts (in *aspectinfo*), and then make appropriate changes to the shadow finder and shadow matcher (in *matching*). More complex extensions may also introduce new kinds of residue, or directly modify the weaving process, but for the examples discussed here, that is not needed.

abc.⟨extension⟩	
parse	new lexer and grammar rules
ast	new ast classes
visit	new compiler passes
extension	overrides of existing ast behaviour
types	new types in typechecker
weaving	
aspectinfo	new IR for pointcuts
matching	finding new shadows; matching shadows to pointcuts
residues	new residue kinds
weaver	changes to the weaver

Fig. 4. Package structure of *abc* extensions.

Driver classes Apart from extending the packages in Figure 4, an extension author must bind all the new functionality together, so that it can be invoked (via reflection) by the base compiler. There are two 'driver' classes for this purpose in *abc*, which any extension must subclass.

The first of these is the *AbcExtension* class. An extension can be specified when abc is invoked by passing its core package name to *abc* with the *-ext* flag. The *AbcExtension* class from this package is then loaded by reflection. All the extensibility hooks in *abc* are passed through this class. There is a default implementation of this class in the *abc.main* package, which extensions must subclass.

Another driver class is *ExtensionInfo*. This is part of the extensibility mechanism of Polyglot; all frontend extensions (except for the lexer) are registered by subclassing this class. New instances of this class are returned by the subclassed *AbcExtension*.

Runtime Some extensions need support in the AspectJ runtime. Indeed, to access reflective information about a new type of joinpoint, we need to make sure the runtime is extended, so this is usually the last step required in implementing a new extension. We shall discuss a concrete example in Section 5.8.

Sources of extensibility It may be helpful to point out at this stage what extensibility is unique to *abc*, and what extensibility has been inherited from the components we built on. We now briefly discuss that, going through the packages in Figure 4. Polyglot provides syntax extensibility; we have added an extensible lexer in *abc*. The way AST nodes are extended in *abc* is based on the principles of Polyglot. Of course the specific interfaces, say for implementing pointcuts, are unique to *abc*. Furthermore, more than half of the passes in *abc* are specific to AspectJ, and therefore the extensibility for introducing new aspect features is in large part determined by our design for those passes. The very small number of overrides of existing AST classes in Polyglot (in the *extension* package) is testament to the extensibility of Polyglot's Java compiler itself. All parts of the weaver are particular to *abc*, although (as further discussed in Section 6) it shares a lot of common structure with *ajc*. A particular feature that enables the extensibility of *abc*'s weaver is the use of the Jimple intermediate representation. Because this is so much easier to analyse and manipulate than either Java source or bytecode, extenders will find it much easier to implement crucial components like a new shadow matcher.

5.2 Extending the lexer

As described in Section 3.1, *abc*'s lexer is stateful. There are four main lexer states for dealing with the different sub-languages of AspectJ: JAVA, ASPECTJ, POINTCUT and POINTCUTIFEXPR. The first three are used in Java code, AspectJ code and pointcut expressions, respectively. The POINTCUTIFEXPR state must be separate from the normal JAVA state because the **if** pointcut allows a Java expression to be nested inside a POINTCUT, but whereas the JAVA state is terminated by a '}', we need to return to the POINTCUT state when reaching a matching closing ')' character.

Keywords for each state are stored in state-specific *HashMap*s which map each keyword to an object implementing the *LexerAction* interface. This interface declares a method

```
public int getToken(AbcLexer lexer)
```

which is called when the corresponding keyword is recognised. Its return value is turned into a parser token and passed to the parser for further analysis. A reference to the lexer instance is passed as a parameter to *getToken*(...), so that side effects that affect the lexer (like changing the lexer state) are possible. A default implementation of this interface is supplied, which offers sufficient functionality to associate keywords with parser tokens and (optionally) change the lexer state; custom implementations of *LexerAction* can provide more flexibility. The default implementation provides functionality sufficient for all but 5 (out of more than 90) Java and AspectJ keywords.

Implementing the *eaj* extensions required adding several new keywords. In particular, "*cast*" was introduced as a keyword in the POINTCUT state, and "*global*" as a keyword in all four lexer states. Both "*private*" and "*throw*" are already keywords in all states, and so do not need to be introduced specifically for the private pointcut variables and throw pointcut extensions. Here is the code that adds the keywords to the respective states:

```
public void initLexerKeywords(AbcLexer lexer)
{
    // keyword for the "cast" pointcut extension
    lexer.addPointcutKeyword("cast",
        new LexerAction_c(new Integer
            (abc.eaj.parse.sym.PC_CAST)));

    // keyword for the "global pointcut" extension
    lexer.addGlobalKeyword("global",
        new LexerAction_c(new Integer
            (abc.eaj.parse.sym.GLOBAL),
                new Integer(lexer.pointcut_state())));

    // Add the base keywords
    super.initLexerKeywords(lexer);
}
```

Both keywords use the default implementation of *LexerAction*, i.e. the *LexerAction_c* class. We see the one-argument and two-argument constructors for that class. The first argument is always the parser token that should be returned for the keyword; the second argument (if present) is the lexer state that should be selected after the keyword. As stated above, further logic can be implemented by subclassing *LexerAction_c*.

5.3 Extending the parser

The grammar fragment below shows how two new productions are added for private pointcut variables and the cast pointcut, which can appear anywhere a normal pointcut could:

```
extend basic_pointcut_expr ::=
      PRIVATE:x LPAREN formal_parameter_list_opt:a RPAREN
      LPAREN pointcut_expr:b RPAREN:y
      {:
            RESULT =
                  parser.nf.PCLocalVars(parser.pos(x,y), a, b);
      :}
|     PC_CAST:x LPAREN type_pattern_expr:a RPAREN:y
      {:
            RESULT =
                  parser.nf.PCCast(parser.pos(x,y), a);
      :}
;
```

The fragment closely resembles code one would use with the popular CUP parser generator, apart from the **extend** keyword, which signifies that these two productions are to be added to the rules that already exist for the nonterminal symbol *basic_pointcut_expr*.

The first new production is for private pointcut variables. As will be apparent from this example, terminal tokens are indicated by capitals. Note that it is possible to bind the result of parsing each grammar symbol to an identifier, indicated by a colon and a name. For instance we bind the result of recognising the token PRIVATE to x, and the result of recognising a *pointcut_expr* to b. These named results can then be used in the parser action associated with a production. This action is delineated with curly braces and colons. Here we use the results of the first and last symbol in the right-hand side of the production to compute the position (via the call $parser.pos(x,y)$) of the whole private pointcut variable declaration. Positions in Polyglot are always a start location (source file, line number, column number) together with an end location. Throughout *abc*, great care is taken to preserve such position information, so that it is possible to track the origin of every piece of code, even after optimisations have been applied. The second grammar production in the above code fragment is for cast pointcuts, and as it is simpler than the first production, we do not discuss it further.

Apart from extending the alternatives for existing nonterminals (as we did above), the Polyglot Parser Generator PPG [9] also allows you to drop productions, transfer productions from one non-terminal to another, and override the productions of a particular non-terminal.

5.4 Adding new AST nodes

As mentioned above, *abc*'s frontend is built on the Polyglot extensible compiler framework [37]. In fact, from Polyglot's point of view, *abc* is just another extension. This means that *abc* "inherits" all the extensibility mechanisms provided by Polyglot.

In particular, adding new AST nodes is common when writing compiler extensions, and thus it is important to provide an easy and robust mechanism for doing so.

All four extensions discussed above required new AST nodes. For the sake of brevity we will only present the node introduced by the global pointcut extension here — the other cases are handled very similarly.

In order to write a clean Polyglot extension, one has to adhere to the rigorous use of factories and interfaces to create nodes and invoke their members, respectively. The first step is therefore to define an interface for the new AST node, declaring any functionality it wants to present to the outside world:

```
public interface GlobalPointcutDecl extends PointcutDecl
{
    public void registerGlobalPointcut(GlobalPointcuts visitor,
                                       Context context,
                                       EAJNodeFactory nf);
}
```

We provide a method to insert the pointcut into a static data structure keeping track of the global pointcuts defined in the program (*cf.* Section 5.5). Note that the interface extends *abc*'s *PointcutDecl* interface, so it provides all the functions relevant to a pointcut declaration.

The next step is to write the class implementing that interface. Some boilerplate code is required (a constructor and methods to allow visitors to visit the node), and, of course, the method *registerGlobalPointcut*() is given a concrete implementation.

In order to make sure we can instantiate this new node type, we subclass *abc*'s default node factory (which, in turn, is derived from Polyglot's node factory) and create a method for obtaining an instance of *GlobalPointcutDecl*:

```
public GlobalPointcutDecl
        GlobalPointcutDecl (
                Position pos,
                ClassnamePatternExpr aspect_pattern,
                Pointcut pc, String name,
                TypeNode voidn )
{
    return new GlobalPointcutDecl_c(pos, aspect_pattern,
                                    pc, name, voidn);
}
```

Now the extended parser can produce *GlobalPointcutDecl* objects when it encounters the appropriate tokens (*cf.* listing in Section 5.3).

Note that all changes are local to new classes we created (in fact, these classes are in a completely separate package). The fact that *abc* itself didn't have to be changed at all makes the extension robust with respect to *abc* upgrades. Also, since the new AST node extends an existing node, very little functionality needs to be re-implemented. The associated interfaces only have to declare the methods specific to the new node's particular functionality.

In the same way, interfaces *PCLocalVars* and *PCCast* were defined, along with implementing classes, for the private pointcut variables and **cast** pointcut extensions. Corresponding factory methods were added to the extended AspectJ node factory.

5.5 Adding new frontend passes

Implementing the "global pointcuts" extension described in Section 4.2 requires two new passes — first, all global pointcuts need to be collected, and then each pointcut must be replaced with the conjunction of the original pointcut and all applicable global pointcuts.

Polyglot's visitor-based architecture makes implementing this very easy. We add two new passes. The first stores all global pointcuts in a static variable, and the second applies that pointcut to the relevant code. For reasons of code brevity, these two passes are implemented by the same class, *GlobalAspects*; it uses a member variable called *pass* to distinguish which of the two functions it is performing.

The traversal of the AST is performed by the *ContextVisitor* Polyglot class. The new pass extends *ContextVisitor* with a method that performs the required action when it encounters a relevant AST node.

The following code fragment illustrates the behaviour of the new visitor upon entering an AST node:

```
public NodeVisitor enter(Node parent, Node n) {
    if (pass == COLLECT
      && n instanceof GlobalPointcutDecl) {
        ((GlobalPointcutDecl) n).
          registerGlobalPointcut(this, context(), nodeFactory);
    }
    return super.enter(parent, n);
}
```

As mentioned above, both new passes are implemented by the same class, and hence the check that *pass==COLLECT* makes sure that we do the right thing. If the current node is a *GlobalPointcutDecl* (one of the new AST nodes defined in section 5.4), we call its special method so it registers itself with the data structure storing global pointcuts. Then we delegate the rest of the work (the actual traversal) to the superclass.

The implementation of the *leave*() method, which is called when the visitor leaves an AST node and has the option of rewriting the node if necessary, is very similar. If *pass==CONJOIN* and we are at an appropriate node, we return the conjunction of the node and the global pointcut.

The sequence of passes that the compiler goes through is specified in the special singleton *ExtensionInfo* class. By subclassing it and inserting our new passes in an overridden method which then calls the original method, we make sure the original sequence of passes is undisturbed. Note that this mechanism makes the extension robust with respect to changes in the base *abc* passes — we can add and rearrange passes without breaking the extension.

5.6 Adding new join points

To implement the cast and throw pointcuts, we first need to extend the list of join point types. This is done by adding to a list of factory objects which the pointcut matcher iterates over to find all join point shadows. The *listShadowTypes* method is defined in the *AbcExtension* class and is overridden for *eaj*: (here and elsewhere, the element type of a collection is indicated by a comment of the form */*<ShadowType>*/*)

```
protected List /*<ShadowType>*/ listShadowTypes()
{
    List /*<ShadowType>*/ shadowTypes =
            super.listShadowTypes();
    shadowTypes.add(CastShadowMatch.shadowType());
    shadowTypes.add(ThrowShadowMatch.shadowType());
    return shadowTypes;
}
```

The definitions of *CastShadowMatch* and *ThrowShadowMatch* are very similar and we therefore limit ourselves to discussing the former.

The *CastShadowMatch.shadowType*() method just returns an anonymous factory object which delegates the work of finding a join point to a static method in the *CastShadowMatch* class. This method, *matchesAt*(...), takes a structure describing a position in the program being woven into and returns either a new object representing a join point shadow or *null*; the code for it is given in Figure 5.

```
public static CastShadowMatch
                matchesAt(MethodPosition pos)
{
    if (!(pos instanceof StmtMethodPosition))
        return null;

    Stmt stmt = ((StmtMethodPosition) pos).getStmt();

    if (!(stmt instanceof AssignStmt))
        return null;
    Value rhs = ((AssignStmt) stmt).getRightOp();

    if (!(rhs instanceof CastExpr))
        return null;
    Type cast_to = ((CastExpr) rhs).getCastType();

    return new CastShadowMatch(
                pos.getContainer(), stmt, cast_to);
}
```

Fig. 5. The CastShadowMatch.matchesAt(...) method

The purpose of the *MethodPosition* parameter is to allow *abc* to iterate through all the parts of a method where a join point shadow can occur, and ask each factory object whether one actually does. There are four types of *MethodPosition* for normal AspectJ shadows:

- Whole body shadows: execution, initialization, preinitialization
- Single statement shadows: method call, field set, field get
- Statement pair shadows: constructor call
- Exception handler shadows: handler

Most shadows either fall into the category of "whole body" or "single statement". Two are special, namely constructor call join points and handler joinpoints. In both cases, the special nature derives from the representation of their shadows in Java bytecode, and consequently their representation in Jimple. In Java bytecode, a constructor call is not a single instruction, but instead it consists of two separate instructions: *new* creates a new instance, whereas *invokespecial* initialises it. A constructor call join point therefore encompasses both of these instructions. Handler join points can only be found by looking at the exception handler table for a method, rather than its statements.

If a new join point requires an entirely new kind of method position, then the code that iterates over them can be overridden.

The first job of the *matchesAt*(...) method is to check that we are at the appropriate position for a **cast** pointcut, namely one with a single statement. Next, we need to check whether there is actually a **cast** taking place at this position; the grammar of Jimple makes this straightforward, as a **cast** operation can only take place on the right-hand side of an assigment statement. If no such operation is found, we return *null*; otherwise we construct an appropriate object.

Defining the *CastShadowMatch* class also requires a few other methods, connected with defining the correct values to be bound by an associated **args** pointcut, reporting the information required to construct a *JoinPoint.StaticPart* object at runtime, and recording the information that a pointcut matches at this shadow in an appropriate place for the weaver itself to use. The details are straightforward, and we omit them for reasons of space.

5.7 Extending the pointcut matcher

Again, we describe the implementation of the **cast** pointcut and omit discussion of the almost identical **throw** pointcut. Once the corresponding join point shadow has been defined, writing the appropriate backend class is straightforward. The pointcut matcher tries every pointcut at every join point shadow found, so all the **cast** pointcut has to do is to check whether the current shadow is a *CastShadowMatch*, and if so verify that the type being cast to matches the *TypePattern* given as argument to the **cast** pointcut:

```
protected Residue matchesAt(ShadowMatch sm)
{
    if (!(sm instanceof CastShadowMatch))
        return null;
    Type cast_to = ((CastShadowMatch) sm).getCastType();

    if (!getPattern().matchesType(cast_to))
        return null;

    return AlwaysMatch.v();
}
```

The *AlwaysMatch.v*() value is a *dynamic residue* that indicates that the pointcut matches unconditionally at this join point. For those pointcuts where matching cannot be statically determined, this is replaced by one which inserts some code at the shadow to check the condition at runtime.

5.8 Extending the runtime library

AspectJ provides dynamic and static information about the current join point through *thisJoinPoint* and associated special variables.

For the **cast** pointcut extension, this runtime interface was extended to reveal the signature of the matching cast. For example, the following aspect picks out all casts (except for the one in the body of the advice) and uses runtime reflection to display the type that is being cast to at each join point:

```
import org.aspectbench.eaj.lang.reflect.CastSignature;

aspect FindCasts
{
    before():
        cast(*) && !within(FindCasts)
    {
        CastSignature s = (CastSignature)
            thisJoinPointStaticPart.getSignature();

        System.out.println("Cast to: " +
                s.getCastType().getName());
    }
}
```

Implementing this requires changes both in the backend of the compiler (where the static join point information is encoded for the runtime library to read later), and the addition of new runtime classes and an interface.

Static join point information is encoded in a string which is parsed at runtime by a factory class to construct the objects accessible from *thisJoinPointStaticPart*. This happens just once, namely in the static initialiser of the class where the join point shadow is located. The alternative, which is to directly generate code to construct these objects, would be expensive in terms of the size of the bytecode produced; using strings provides a compact representation without too much runtime overhead.

The static information for a **cast** pointcut is encoded as follows. To allow us to easily reuse the existing parser for such strings, a fair amount of dummy information is generated, corresponding to properties that cast join points do not have. For example, modifiers such as **public** are important for join points that have a method or field signature associated with them, but make no sense for the cast join point. The string for the **cast** pointcut is constructed from four parts:

– Modifiers (encoded as an integer — 0 for a cast)
– Name (usually a method or field name, but for a cast it is just "cast")
– Declaring type — class in which the join point occurs

– Type of the cast

For example, a cast join point within a method in the class *IntHashTable* which casts the value retrieved from a *HashMap* to an *Integer* would produce the following encoded string:

```
"0-cast-IntHashTable-Integer"
```

The runtime factory is subclassed to add a method that creates an object implementing the new *CastSignature* interface for appropriate join points. The aforementioned *AbcExtension* class has a method which specifies which runtime class should be used as a factory for **thisJoinPointStaticPart** objects, which is overriden so that runtime objects are created with the new factory:

```
public String runtimeSJPFactoryClass()
{
    return
        "org.aspectbench.eaj.runtime.reflect.EajFactory";
}
```

5.9 Code measurements

To enable the reader to assess the amount of effort involved in implementing each of these new features, we have summarised some statistics in Figure 6. The table shows the size of the whole parser, and of the boilerplate for factories in the top and penultimate row, respectively. The most interesting part is the breakdown by construct in the middle. For private pointcut variables, all the work goes into defining new AST nodes, and there is no need to define new passes or to touch the weaver in any way. By contrast, global pointcuts require the introduction of new Polyglot passes, which reduce the new construct to existing AspectJ constructs. Finally, for cast and throw pointcuts, there is substantial work in the weaver, because these introduce a new type of join point.

It is pleasing to us that the distinction between the examples is so sharp, as it gives good evidence that the aim of modularity has been achieved. This claim is also backed up by the fact that none of the extensions required any change to the code of the base compiler: the extensions are clearly separated plugin modules. We believe that the amount of code that needs to be written also meets the criterion of proportionality that was introduced at the beginning of this paper. The criterion of simplicity is more difficult to measure, but we hope that the sample code in this section suffices to convince the reader that we have succeeded in this respect as well. The examples presented here do not demonstrate analysis capability: in Section 7 we do however discuss some more substantial case studies done by others which make essential use of the analysis framework in *abc*.

6 Detailed comparison to *ajc*

The *de facto* standard workbench for research into variations and extensions of AspectJ is the *ajc* compiler. It has served this purpose admirably well, and for example [31, 39] report on the successful integration of substantial new features into *ajc*.

eaj **measurements**		Files	Lines of code
Parsing		1	74
Private pointcut variables	AST nodes	2	130
	Passes	0	0
	Weaver	0	0
	Runtime	0	0
Global pointcut declarations	AST nodes	4	64
	Passes	1	77
	Weaver	0	0
	Runtime	0	0
Cast poincuts	AST nodes	2	46
	Passes	0	0
	Weaver	2	94
	Runtime	2	27
Throw pointcuts	AST nodes	2	46
	Passes	0	0
	Weaver	2	91
	Runtime	2	16
Extension information and shared classes		7	205
Total		27	870

Fig. 6. Code measurements for *eaj*

Throw-pointcut statistics	*ajc*	*abc*
Core compiler/runtime files modified	8	0
throw-specific files created	2	6
Factory subclasses created	-	5
Total files touched	10	11
Lines of code written[1]	103	187

Fig. 7. The **throw** pointcut in *ajc* and *abc*.

We believe that, in view of the explosion of research into new features and analyses, the time has now come to disentangle the code of the base compiler from that of the extensions. The benefits are illustrated by the table in Figure 7. It compares the implementation of the **throw** pointcut in *abc* and *ajc*. In the case of *ajc*, we have to modify a large number of existing files, thus tangling the new extension with the existing compiler base. At the cost of some subclassed factories (and thus some more lines of code), *abc* disentangles the two completely: there is no need to modify any part of the base code, and *abc* extensions are clearly separated plugin modules.

These differences follow directly from the design goals of *ajc*, which 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 [27]. By contrast, *abc*'s overrid-

[1] Note that the numbers in Figure 7 for *abc* take into account the relevant lines of files which are listed under "Extension information and shared classes" in Figure 6.

ing design goals are extensibility and optimisation, as well as a complete separation from the components it builds on. In the remainder of 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 the 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 have complex dependencies. 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 nodes 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 inquire 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 AspectJ 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 six benchmarks from [16], as shown in Table 8. Our experiments were done on a dual 3.2GHz Xeon with 4GB 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. The first column shows the benchmark name. We then give the size of the source in lines (as counted with *sloccount*) and the number of times advice needs to be woven into a

benchmark	SLOC	APPS	*ajc*	*ajc + soot*	*abc-O0*	*abc*	*javac*
bean	124	4	1.77	4.00	3.30	3.59	-
bean-java	104	0	1.43	3.21	3.05	3.03	0.54
sim-nullptr	1474	138	2.96	12.00	10.38	10.69	-
sim-nullptr-java	1547	0	1.75	6.52	7.45	8.64	0.76
figure	94	12	1.62	3.43	2.95	3.07	-
figure-java	98	0	1.25	2.83	2.63	2.65	0.51
LoD-sim	1586	1332	4.10	29.87	36.47	46.14	-
dcm	1668	359	3.37	17.07	14.74	17.43	-
tetris	1043	29	2.88	8.42	8.40	8.93	-

Fig. 8. Compile times using *ajc*, *abc* and *javac* (seconds)

shadow. The remainder of the columns show the four different compilers, plus *javac* where applicable.

The first three AspectJ benchmarks (bean,figure,sim-nullptr) have Java equivalents, where the weaving has been performed by hand (bean-java,figure-java,sim-nullptr-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 three benchmarks (LoD-cflow, dcm, tetris) make heavy use of aspects so there are no hand-woven Java equivalents.

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 sim-nullptr benchmark is typical: the difference between *abc* and *ajc* for programs of a few thousand lines is usually a factor of about 4. For examples where *abc* does a lot of optimisation, such as LoD-sim, the gap can be slightly larger. 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 quite 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*)

Fig. 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 at 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, 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*. In earlier work, in preparation for the construction of *abc* itself, we conducted a detailed study of the dynamic behaviour of aspect-oriented programs [16]. Through a specially constructed set of measurement tools, we were able to confirm the common belief that in many AspectJ programs the overhead introduced by aspects is negligible. However, we were also able to identify common cases where the overheads are surprisingly high. Motivated by these results, we made it an explicit goal of *abc* to be able to experiment with new aspect-specific optimisations.

Because optimisations are an explicit design goal of *abc*, it is important that such experiments are thorough and realistic. In a companion paper [7], we provide a detailed account of the most important optimisations in *abc*, and of their effect on run times. The reader is referred to that paper for a detailed technical account aimed at compiler writers; below we review the most salient points that are relevant to the present comparison with *ajc*.

The first kind of optimisation is an improved implementation of **around** advice, giving a 6-fold speedup of on some benchmarks. In certain cases, *ajc* reverts to generating closures in order to implement **proceed**. When this happens, a lot of heap space is used, leading to very significant overheads. By contrast, in *abc* we are able to avoid the construction of closures in all but very rare pathological cases. In cases where *ajc* does *not* generate closures, it performs a great deal of inlining. This can result in significant code bloat, especially where the advice is woven at many different joinpoint shadows. Again, the compilation strategy employed by *abc* strikes a careful balance between code size and speed. This is illustrated in Figure 10. Further details of the benchmarks can be found in [7, 16].

Benchmark	Time (s) *abc*	Time (s) *ajc*	Size (instr.) *abc*	Size (instr.) *ajc*
sim-nullptr	21.9	21.4	7893	10186
sim-nullptr-rec	23.6	124.0	8216	10724
weka-nullptr	19.0	16.0	103018	134290
weka-nullptr-rec	18.9	45.5	103401	130483
ants-delayed	17.5	18.2	3688	3785
ants-profiler	22.5	21.2	7202	13401

Fig. 10. Execution Times and Code Size

The second kind of optimisation is a set of intraprocedural improvements to **cflow**. In *ajc* 1.2, the implementation of **cflow** used expensive manipulations of a stack, where a simple counter would have sufficed. Also it retrieved the same thread-local state multiple times in a single procedure body, and it did not share work between multiple occurrences of the same **cflow** pointcut. All these problems were eliminated in *abc*, and compared to version 1.2 of *ajc*, these small optimisations yield improvements of 182 $\times$ (the LoD-sim benchmark). The simplest of these optimisations (counters and sharing) were incorporated into *ajc* 1.2.1.

In earlier work, prior to the start of the *abc* project, we showed how an interprocedural analysis can be used to completely eliminate the cost of **cflow** [40]. This is a good example where the full analysis capabilities of *abc* come into play. The essential idea is to construct a static approximation of the dynamic call graph, so that for each shadow, we can determine at compile time whether it will be in the **cflow** of a given pointcut. Such call graph construction is notoriously hard [21], and thus it is important that we do not need to construct a new analysis from scratch for AspectJ, or indeed for every extension of AspectJ.

We would therefore like to leverage existing analyses for pure Java. To that end, *abc* provides the technique of *re-weaving*, which we briefly touched upon in Section 2, in particular Figure 2. The compiler does a first pass over the program, weaving advice naively. The result of this process is a representation of the complete program as pure Jimple code, without any aspect-oriented features. This is then analysed in the usual manner. The results of the analysis are fed back into an optimiser of the *advice-lists*, which can be viewed as little meta-programs that contain instructions to the weaver. The optimisations usually consist of turning a piece of dynamic residue (like updates of the **cflow** stack) into a no-op.

The effectiveness of our optimisations of **cflow** is shown in Figure 11. The message for researchers that wish to implement their own advanced extensions to AspectJ is that *abc* provides the necessary infrastructure to overcome the challenge of implementing these new features efficiently. It is our belief that new proposals for robust semantic pointcuts (*e.g.* [15,45]) necessitate the same type of optimisations and analyses that we have used to make **cflow** efficient.

Benchmark	*abc* no-opt	*abc* sharing	*abc* sharing+ counters	*abc* sharing+ counters+ reuse	*abc* +inter-proc	*ajc* 1.2 (no-opt)	*ajc* 1.2.1 (sharing+ counters)
figure	1072.2	238.3	90.3	20.3	1.96	450.5	167.7
quicksort	122.3	75.1	27.9	27.4	27.3	123.5	28.9
sablecc	29.0	29.1	22.8	22.5	20.4	29.7	24.2
ants	18.7	18.8	18.7	17.9	13.1	33.0	32.9
LoD-sim	1723.9	46.6	32.8	26.2	23.7	4776.2	35.3
LoD-weka	1348.7	142.5	91.9	75.2	66.3	2349.2	113.5
Cona-stack	592.8	80.1	41.2	27.4	23.1	1107.4	56.0
Cona-sim	75.8	75.3	73.8	72.0	73.6	76.8	69.0

Fig. 11. Optimisations of **cflow**

7 Related work

The related work falls into two parts. First of all, others have made an independent assesment of the extensibility of *abc*, by implementing extensions of their own. We first discuss some of these. Second, we review a number of alternative proposals for building an AOP language workbench, and we contrast them with the approach taken in *abc*.

7.1 Users of *abc*

Harbulot and Gurd apply aspect-oriented techniques to parallelise scientific code [25]. For these applications, it is imperative to be able to define joinpoints for loop iteration — the alternative is to refactor the code to expose such join points via spurious method calls. It is, however, not an easy task to define a robust notion of loop join points, which does not depend on the syntactic presentation of the code. This problem is addressed in [26], and solved by a language extension that is implemented in *abc*.

To illustrate, suppose that we wish to advise loop iterations over a given array. Say we want to intercept the loop

```
for (int i = 0; i < array.length; i += 1) {
    Object item = array[i];
    ...
}
```

In the proposed extension of Harbulot and Gurd, this can be achieved with the pointcut

```
pointcut arrit(Object[] array, int min, int max, int stride) :
    loop() && args(min, max, stride, array);
```

Note, however, that it is highly non-trivial to detect the relevant patterns in bytecode; their implementation first recovers loop structure by computing dominators, and then it does a flow analysis of the loop body to determine the loop variable, its lower and upper bound (0 and *array.length* above), as well as the stride (1 in the above example). The joinpoint shadow matching depends on the precision of these analyses: there may be loop iterations for which the correct *min*, *max* and *stride* cannot be statically determined. The implementation described by [26] does however work independent of whether the user employed **while** or **for** to express a computation.

This case study thus provides a good example of the need for strong analysis capabilities in an extensible compiler for AspectJ. Similar examples abound in the literature, such as Kiczales' *predicted cflow*. The analysis capabilities of *abc* are also indispensable to efficiently implement advanced pointcuts such as the *dataflow pointcut* of [31].

Stolz and Bodden propose to use aspect-orientation for the runtime verification of temporal properties. They define an extension of AspectJ where the user can specify properties as LTL formulae [42]. The implementation is an extension of *abc*.

The atoms of the LTL formulae are pointcuts; and a formula as a whole is translated into an alternating automaton, coded as a regular AspectJ aspect. The translation is thus done entirely using Polyglot, and no changes to the backend are needed. This illustrates one of the advantages of our architecture: it has a gentle learning curve, and there is no need to enter into the complications of generating Jimple if that is not desired.

Experience seems to suggest that many beginning users of *abc* start by implementing an extension as a source-to-source transformation very early on in the compiler, even prior to name disambiguation. Then, when more sophisticated error checking is required, the transformation is moved later, and delayed until all checking is complete. Indeed, such is the intended use of the Polyglot framework.

In the case of these novel features for runtime verification, however, there would be a clear benefit to delaying at least part of the code generation even further, so that it is possible to take advantage of the analysis framework in the backend to examine control flow. Again, *abc* provides all support necessary for making such a step from the implementation described in [42].

Aotani and Masuhara It is natural to seek language-level mechanisms to enhance the expressive power of pointcuts. A particularly promising approach is put forward by Aotani and Masuhara [3], and they have implemented it with *abc*. Here the idea is to use **if**-pointcuts and joinpoint reflection to conveniently express pointcuts such as "all calls where the declared type of the receiver is an interface":

```
pointcut interfaceCall() :
    call(* *(..)) && if(isInterface(thisJoinPoint));

static boolean isInterface(JoinPoint tjp) {
    return tjp.getSignature().getDeclaringType().isInterface();
}
```

When used directly in AspectJ, this would lead to quite inefficient code.

Instead, Aotani and Masuhara adopt the perspective of *partial evaluation*, evaluating **if** pointcuts at compile time. Strictly speaking this is therefore not an extension of the AspectJ language, but rather a change in compilation strategy. Again both the Polyglot-based frontend and the Soot-based backend lend themselves very well to implementing such transformations.

Other extensions of abc The overview above is not exhaustive, and many other researchers are actively developing extensions of *abc*. Examples include DJCutter (a distributed AOP language) [36]; Cona (a tool for checking contracts) [41]; trace-based aspects [15, 45]; a model checker for aspects [30]; and tools to perform tasks such as slicing [46]. We are very encouraged by all these developments, and we believe it provides fairly strong independent evidence of the claims for *abc*'s extensibility made in this paper.

7.2 Other workbenches for AOP language research

Of course we are not the first to realise the need for a workbench to conduct aspect-oriented programming language research, and below we review some earlier approaches put forward by others.

Javassist Javassist is a reflection-based toolkit for developing Java bytecode translators [11]. Compared to other libraries such as BCEL, it has the distinguishing feature that transformations can be described using a source-level vocabulary. Compared to *abc*, it provides some of the combined functionality of the Java-to-Jimple translator plus the advice weaver, but its intended applications are different: in particular, it is intended for use at load-time. Consequently, Javassist does not provide an analysis framework like Soot does in *abc*. In principle, such a framework could be added, but it would require the design of a suitable intermediate representation akin to Jimple.

Josh Josh is an open implementation of an AspectJ-like language based on Javassist [10], and as such it is much closer in spirit to *abc*. Indeed, the primary purpose of Josh is to experiment with new pointcut designators, although it can also be used for features such as parametric introductions. Because of the implementation technology, there is no special support for the usual static checks in the frontend, which is provided in *abc* by the infrastructure of Polyglot. Josh does not cover the whole of AspectJ, which limits its utility in realistic experiments.

Logic meta-programming A more radical departure from traditional compiler technology is presented by *logic meta-programming*, as proposed by [13, 22]. Here, program statements where extra code should be woven in are selected by means of full-fledged Prolog programs. This adds significant expressive power, and like Josh, the design makes it easy to experiment with new kinds of pointcuts. The system operates on abstract syntax trees, which are not a convenient representation for transformation and analysis — many years of research in the compilers community have amply demonstrated the merits of a good intermediate representation. A further disadvantage, in our view, is the lack of static checks due to the increased expressive power. The success of AspectJ can partly be explained by the fact that it provides a *highly disciplined* form of meta-programming; some of that discipline is lost in logic meta-programming, because the full power of Prolog precludes certain static checks. Nevertheless, a system based on these ideas is publicly available [44], and it is used as a common platform by a number of researchers.

Pointcuts as functional queries Eichberg, Mezini and Ostermann have very recently suggested an open implementation of pointcuts, to enable easy experimentation with new forms of pointcuts [18]. Their idea is closely related to that of logic meta-programming, namely to use a declarative query language to identify join point shadows of interest. A difference is that they opt for the use of the XML query language XQuery instead of a logic language. Furthermore, [18] only deals with static join points. As argued in the introduction, several recent proposals for new pointcut primitives require data flow analyses. We believe that it is not convenient to express such analyses via queries on syntax trees. It is however quite easy to transfer some of the ideas of [18] to *abc*, by letting the queries range over Polyglot ASTs. A challenge, then, is to define appropriate type rules to implement as part of the frontend.

8 Conclusions and Future Work

We have presented *abc*, and its use as a workbench for experimentation with extensions of AspectJ. Our primary design goal was to completely disentangle new features from the existing codebase, and this goal has been met. In particular, extensions need not make any changes to the code of the base compiler: they are truly separated plugin modules. We hope that such disentangling will enable yet more rapid developments in the design of aspect-oriented programming languages, and the integration of ideas from multiple research teams into a single system, where the base can evolve independently of the extensions.

This project has also been an evaluation of the extensibility of Polyglot and Soot, from the perspective of aspect-oriented software development. We now summarise their role in the extensibility of our design, and identify possible improvements.

Polyglot Polyglot turned out to be highly suited to our purposes. Its extension mechanisms are exactly what is needed to implement AspectJ itself as an extension of Java, with only minimal code duplication. This in turn makes the development of *abc* relatively independent of further improvements to Polyglot.

As we have remarked earlier, the Polyglot mechanism of *delegates* mimicks that of ordinary intertype declarations, whereas *extension nodes* roughly correspond to what an AspectJ programmer would naturally do via **declare parents** and interface intertype declarations. Polyglot achieves this effect by cunningly creating a replica of the inheritance hierarchy in code, which then provides the hooks for appropriate changes. Arguably that mechanism is somewhat brittle, and it is certainly verbose, replicating the same information in multiple places of the code.

We thus face the question whether it would be possible to extend *abc* using AspectJ, or indeed any other dialect of Java that features open classes. The answer is in the positive, as *abc* is written in pure Java. Todd Millstein has used Relaxed MultiJava [35] in precisely this way, using open classes in lieu of Polyglot's delegate and extension nodes, to implement his recent work on predicate dispatch [34]. It follows that users who prefer to use AspectJ to extend *abc* can do so without further ado.

Would the result be more compact and understandable code? Unfortunately, a significant proportion of Polyglot's extensions is taken up by boilerplate code for generic visitors in each new AST node. To generate that automatically, one would need reflection or a feature akin to parametric introductions [23]. The reflection route has been used with much success, in a framework by Hanson and Proebsting [24] that is very similar to Polyglot.

On the whole we feel our choice of Polyglot has been justified. To further assess its merits, we are now engaged in a comparative study of Polyglot's extension mechanism and more advanced technologies such as aspect-oriented reference attribute grammars [19]. In particular, we would like to investigate how multiple, independent extensions can be composed.

Soot The choice of Soot as the basis for our code generation and weaver has had a fundamental impact not only on the quality of the code that is generated, but also on the ease by which the transformations are implemented. The Jimple intermediate representation of Soot has been honed on a great variety of optimisations and analyses before we applied it to *abc*, and we reap the benefits of this large body of previous work.

Equally important has been the use of the Dava decompiler that is part of the Soot framework. This makes it much easier to pinpoint potential problems, and to communicate the ideas about code generation to others. It also opens the way to exciting new visualisations, for example to indicate at source level exactly what dynamic residue was inserted at a join point shadow.

In the comparison with *ajc* we demonstrated the importance of the analysis framework in Soot: it is indispensable to eliminate the overheads of advanced language features such as **cflow**. The need for such optimisation is likely to increase with new proposed extensions such as predicted control flow [29], data flow pointcuts [31] and trace cuts [14, 45]. Apart from optimisation, Soot's analysis capabilities are also crucial in the robust implementation of new pointcuts, for instance those for loop iteration [26].

In summary, we have demonstrated (both through experiments of our own and by reviewing work of others) that *abc* provides an extensible framework for experiments in the design of aspect-oriented programming languages, meeting the criteria of *simplicity*, *modularity*, *proportionality* and *analysis capability* set out in the introduction. The next step in its development, namely the upgrade to Java 1.5, will provide a further opportunity to hone these characteristics. Soot is ready for this transition, but Polyglot still needs to be updated to Java 1.5.

Acknowledgments

This work was supported, in part, by NSERC in Canada and EPSRC in the United Kingdom. Our thanks to Chris Allan for his comments on a draft of this paper. Adrian Colyer gave helpful advice on how to collect relevant statistics regarding the source of *ajc*.

References

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

2. Jonathan Aldrich. Open Modules: Modular Reasoning about Advice. In Andrew Black, editor, *19th European Conference on Object-Oriented Programming (ECOOP 2005)*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer-Verlag, 2005.
3. Tomoyuki Aotani and Hidehiko Masuhara. Compiling conditional pointcuts for user-level semantic pointcuts. In *Proceedings of the SPLAT workshop at AOSD 2005*, 2005. Available from `http://www.daimi.au.dk/~eernst/splat05/`.
4. AspectJ bug database. Wrong variable binding in || pointcuts. See `https://bugs.eclipse.org/bugs/show_bug.cgi?id=61568`, 2004.
5. AspectJ bug database. ITD on inner class: missing accessor method. See `https://bugs.eclipse.org/bugs/show_bug.cgi?id=73856`, 2005.
6. Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. *abc*: An extensible AspectJ compiler. In Peri Tarr, editor, *4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, pages 87–98. ACM Press, 2005.
7. Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In Vivek Sarkar and Mary W. Hall, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 117–128. ACM Press, 2005.
8. Jonas Bonér. AspectWerkz — dynamic AOP for Java. Available from URL: `http://codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf`, 2004.
9. Michael Brukman and Andrew C. Myers. PPG: a parser generator for extensible grammars, 2003. Available at `www.cs.cornell.edu/Projects/polyglot/ppg.html`.
10. Shigeru Chiba and Kiyoshi Nakagawa. Josh: an open AspectJ-like language. In Karl Lieberherr, editor, *3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 102–111, 2004.
11. Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In Frank Pfenning and Yannis Smaragdakis, editors, *2nd International Conference on Generative Programming and Component Engineering (GPCE '03)*, volume 2830 of *Lecture Notes in Computer Science*, pages 364–376. Springer-Verlag, 2003.
12. Adrian Colyer and Andrew Clement. Large-scale AOSD for middleware. In Karl Lieberherr, editor, *3rd International Conference on Aspect-Oriented Software Development AOSD 2004*, pages 56–65. Association for Computing Machinery, 2004.
13. Kris de Volder. Aspect-oriented logic meta-programming. In Pierre Cointe, editor, *2nd International Conference on Meta-level Architectures and Reflection*, volume 1616 of *Lecture Notes in Computer Science*, pages 250–272. Springer-Verlag, 1999.
14. Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Karl Lieberherr, editor, *3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 141–150. ACM Press, 2004.
15. Rémi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In Robert Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*. Addison-Wesley, 2004.
16. Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *Proceedings of the 19th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 150–169. ACM Press, 2004.
17. Chris Dutchyn, Gregor Kiczales, and Hidehiko Masuhara. Tutorial: AOP language exploration using the Aspect Sand Box. In Gregor Kiczales, editor, *1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*. ACP Press, 2002.

18. Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In Wei-Ngan Chin, editor, *Second ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *Lecture Notes in Computer Science*, pages 366–381. Springer-Verlag, 2004.
19. Torbjörn Ekman and Görel Hedin. Rewritable reference attributed grammars. In Martin Odersky, editor, *18th European Conference on Object-Oriented Programming (ECOOP 2004)*, volume 3086 of *Lecture Notes in Computer Science*, pages 144–169. Springer-Verlag, 2004.
20. Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In Jens Palsberg, editor, *Static Analysis Symposium*, pages 199–219, 2000.
21. David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In Toby Bloom, editor, *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 108–124. ACM Press, 1997.
22. Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In Mehmet Akşit, editor, *2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 60–69. ACM Press, 2003.
23. Stefan Hanenberg and Rainer Unland. Parametric introductions. In Mehmet Akşit, editor, *2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 80–89. ACM Press, 2003.
24. David Hanson and Todd Proebsting. A research C# compiler. *Software — Practice and Experience*, 34(13):1211–1224, 2004.
25. Bruno Harbulot and John R. Gurd. Using AspectJ to separate concerns in parallel scientific Java code. In Karl Lieberherr, editor, *3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 122–131. ACM Press, 2004.
26. Bruno Harbulot and John R. Gurd. A join point for loops in AspectJ. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *Foundations of Aspect-Oriented Languages (FOAL 2005)*, pages 11–20, 2005. Technical report 05-05, Department of Computer Science, Iowa State University. Available from: `http://www.cs.iastate.edu/~leavens/FOAL/index-2005.shtml`.
27. Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In Karl Lieberherr, editor, *3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 26–35. ACM Press, 2004.
28. Jim Hugunin. Guide for developers of the AspectJ compiler and weaver, 2004. Available at `http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/org.aspectj/modules/docs/developer/compiler-weaver/index.html?rev=1.1&content-type=text/html&cvsroot=Technology_Project`.
29. Gregor Kiczales. The fun has just begun. Keynote address at AOSD. Available at `aosd.net/archive/2003/kiczales-aosd-2003.ppt`, 2003.
30. Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. In Richard N. Taylor and Matthew B. Dwyer, editors, *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 137–146, 2004.
31. Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In Atsushi Ohori, editor, *1st Asian Symposium on Programming Languages and Systems*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121. Springer-Verlag, 2003.
32. Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In Görel Hedin, editor, *12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

33. Jerome Miecnikowski and Laurie J. Hendren. Decompiling java bytecode: problems, traps and pitfalls. In R. Nigel Horspool, editor, *11th International Conference on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 111–127. Springer-Verlag, 2002.
34. Todd Millstein. Practical predicate dispatch. In John M. Vlissides and Douglas C. Schmidt, editors, *Conference on Object-oriented Programming, Systems, Languages and Applications (OOPSLA 2004)*, pages 345–364. ACM Press, 2004.
35. Todd Millstein, Mark Reay, and Craig Chambers. Relaxed MultiJava: Balancing extensibility and modular typechecking. In Ron Crocker and Guy L. Steel Jr., editors, *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2003)*, pages 224–240. ACM Press, 2003.
36. Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote pointcut — a language construct for distributed AOP. In Karl Lieberherr, editor, *3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 7–15. ACM Press, 2004.
37. Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer-Verlag, 2003.
38. Harold Ossher and Peri Tarr. Hyper/J: multi-dimensional separation of concerns for java. In *22nd International Conference on Software Engineering*, pages 734–737, 2000.
39. Kouhei Sakurai, Hidehiko Masuhara, Naoyasu Ubayashi, Saeko Matsuura, and Seiichi Komiya. Association aspects. In Karl Lieberherr, editor, *3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 16–25. ACM Press, 2004.
40. Damien Sereni and Oege de Moor. Static analysis of aspects. In Mehmet Akşit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 30–39. ACM Press, 2003.
41. Therapon Skotiniotis and David H. Lorenz. Cona: aspects for contracts and contracts for aspects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 196–197, New York, NY, USA, 2004. ACM Press.
42. Volker Stolz and Eric Bodden. Temporal Assertions using AspectJ. In *Fifth Workshop on Runtime Verification (RV'05)*, Electronic Notes in Theoretical Computer Science, Edinburgh, Scotland, UK, 2005. Elsevier Science Publishers.
43. 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 David A. Watt, editor, *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.
44. Kris De Volder. The TyRuBa metaprogramming system. Available at `http://tyruba.sourceforge.net/`.
45. Robert Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *ACM Sigsoft International Symposium on Foundations of Software Engineering (FSE-12)*, pages 159–169, 2004.
46. Jianjun Zhao. Slicing aspect-oriented software. In *10th IEEE Workshop on Program Comprehension*, pages 251–260, 2002.