# Modularity First: A Case for Mixing AOP and Attribute Grammars

Pavel Avgustinov, Torbjörn Ekman, Julian Tibble

Programming Tools Group
University of Oxford
United Kingdom

## ABSTRACT

We have reimplemented the frontend of the extensible AspectBench Compiler for AspectJ, using the aspect-oriented meta-compiler JastAdd. The original frontend was purely object-oriented. Each frontend extends Java with AspectJ and an additional set of pointcuts in a modular fashion. In this paper we give a detailed comparison of both approaches and show a number of advantages of using JastAdd: the implementation is half the size, twice as fast, concerns are better localised, extensions are composable, and computations are automatically scheduled.

JastAdd provides a very constrained form of static AOP where only inter-type declarations and method execution interception are supported. However, additional modularisation mechanisms from the compiler construction community are supported in the form of demand-driven evaluation and attribute grammars. Our implementation would not have benefited from a richer pointcut language, while both demand-drive evaluation and declarative attributes were essential in enabling composable extensions and flexible modularisation.

We believe that the AOP community at large can benefit from acknowledging demand-driven evaluation as an important modularisation mechanism. Also, reference attribute grammars enhance the extensible implementation of graph-based computations that rely on context-sensitive information.

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors—*Compilers*

## General Terms

Languages, Design

## Keywords

Extensible compilers, modularity, separation of concerns, aspect-oriented programming

## 1. INTRODUCTION

Although compiler construction is a very well-developed field, the problem of creating *extensible* compilers still poses great challenges from the point of view of modularisation. In programming languages research, it is quite common to extend a language with new features or constructs, and to add additional analyses. Ideally, it should be possible to make such extensions completely self-contained, and to enable and compose them at will, to allow experimentation with different sets of language features. Moreover, extensions should be modularised internally so that each can be understood in isolation. These requirements put extreme tension on traditional modularisation mechanisms provided by the language used to implement them.

The AspectBench Compiler (*abc*) is an extensible AspectJ compiler intended as a workbench for aspect-oriented language research, and it has been successfully adopted as the basis of implementation for a number of extensions (*e.g.* [2,3,14]). The system is divided into a frontend, taking care of static semantic analysis, and a backend, performing optimisation and aspect weaving. The AspectJ frontend is itself implemented as a modular extension to the Polyglot extensible Java frontend framework [23]. Polyglot is an object-oriented framework based on standard Java which relies on extensible visitor patterns for modular extensibility.

We have reimplemented the frontend using the aspect-oriented meta-compiler tool JastAdd [10]. *Extended AspectJ* (EAJ) is a set of modest language extensions the authors of *abc* developed to demonstrate its extensibility; we have also ported these to the new frontend in order to compare the ease with which new features can be added. In this paper, we give a detailed comparison of both approaches and show a number of advantages of using JastAdd: smaller implementation, faster compilation, composable extensions, enhanced localisation of concerns and automatic scheduling of computations.

Much work has been devoted to the so called *Expression Problem* (thus named by Phil Wadler [30]), which concerns separating a model from computations performed on that model, while still enabling modular extensions to both the model and the computations. To some extent, this is indeed addressed by the fact that all information in a JastAdd-based frontend is stored on AST nodes, and so can be adapted as and when new AST nodes are added. However, in our experience a much harder problem is then how to separate conceptually different computations that interact with each other. The key to such separation in the new frontend was to combine ITDs (inter-type declarations) with demand-driven evaluation and attribute grammars. We believe that the AOP community at large can benefit from acknowledging demand-driven evaluation as an important modularisation mechanism. Attribute grammars, with support for references, can enhance extensibility and modularity of context-sensitive graph-based computations compared to using more traditional AOP techniques.

The contribution of this paper is three-fold:

1. We present a large-scale case study comparing aspect-oriented compiler construction to a visitor-based object-oriented approach.

2. We give a qualitative and quantitative comparison of the merits of various aspect-oriented and compiler-construction modularisation mechanisms. In our experiments a very constrained from of static AOP sufficed, but demand-driven evaluation and declarative attributes were essential for composable extensions and localisation of concerns.

3. By improving the already well-received *abc* compiler with the techniques discussed in this paper, we provide a more flexible research platform for AspectJ language extensions.

The rest of the paper is structured as follows. First we give an overview of the Polyglot and JastAdd based frontends for *abc* in Section 2.1. Then we introduce the various modularisation mechanisms used by Polyglot in Section 2.2 and by JastAdd in Section 2.3. Section 3 contains a detailed comparison of the frontends from the point of view of separation of concerns, size, and performance. We discuss, summarise and generalise our results in Section 4. Related work is presented in Section 5 and we conclude and outline future work in Section 6.

## 2. BACKGROUND

### 2.1 *abc* architecture

This section gives a brief overview of the *abc* AspectJ compiler. As usual in the field of compiler construction, the overall architecture can be roughly divided into a frontend and a backend component, and this paper deals mainly with the former — the original *abc* backend is reused as is. The purpose of the frontend is to parse source code into an abstract syntax tree (AST), and then perform static semantic analyses, *e.g.* name binding, type checking, and class hierarchy wellformedness checking. If no errors are found then it instantiates an intermediate code model used by the backend.

The fronted can conceptually be divided into the following phases: lexing, parsing, static semantic analysis, abstract syntax tree transformations, and backend code model instantiation. Lexing and parsing build an initial AST on which semantic analysis is performed. This clean separation enables us to replace the parser with any kind of parsing technology that can build an AST. We can therefore benefit from recent advances in extensible parsing technology, e.g., [12, 29]. In the rest of this paper we assume that a suitable parsing technology has been used to build an AST and static semantic analysis is about to start. Our current implementation uses LALR parsing and a separate scanner with lexical states. Further details on parsing AspectJ are available in [4, 6].

The backend of *abc* uses the Soot optimisation framework for Java and analyses Jimple, which is a typed 3 address intermediate code model [27]. An additional *AspectInfo* model is used by the weaver in the backend to match pointcuts, weave advice, and weave inter-type declarations. The frontend is thus responsible for instantiating both models by generating Jimple code and populating the AspectInfo structure. A common technique to simplify such generation is to first normalise the AST through a series of tree transformations.

Figure 1 shows a schematic overview of the *abc* compiler. The frontend is based on the Polyglot framework for extensible Java frontends [23]. There is no code generation in Polyglot, but Soot provides the *JavaToJimple* component which translates the attributed AST into Jimple. AspectJ is implemented as a separate language component which extends both Polyglot and JavaToJimple. That component implements the features and checks of AspectJ in the frontend and instantiates the AspectInfo structure. Further language components can then be added to the system, *e.g.* EAJ, which adds more pointcuts to AspectJ. The extension is separated into a frontend, for static semantic analysis, and a backend, providing matching for the new pointcuts.

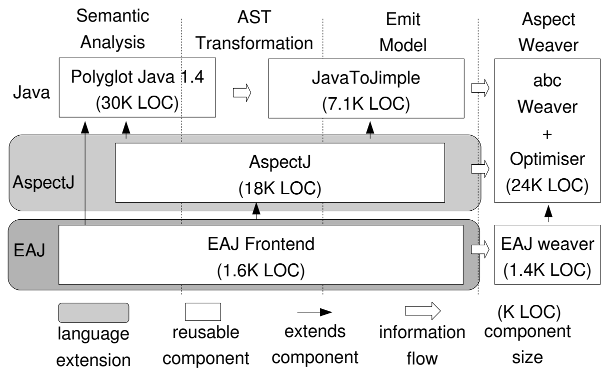

**Figure 1: Overview of the Polyglot-based frontend in *abc*. Components are annotated with their sizes in lines of code. Java 1.4 is modularly extended to support AspectJ and EAJ.**

The clean separation between the frontend and backend allows us to create a replacement frontend which can be plugged into the compiler without changing the backend. Indeed, the new frontend is plugged into *abc* as any other extension with the main difference that it replaces the entire frontend, rather than extending the Polyglot based frontend. That way we can reuse the backend as is without any changes, including the backend extension for EAJ.

Figure 2 shows the new frontend, based on the Java 1.4 frontend in the JastAdd Extensible Java Compiler [10]. JastAdd uses a different AST than Polyglot and we therefore had to use a different component to generate Jimple for the backend. A separate component normalises the AST prior to that generaton, e.g., flattening of nested classes. AspectJ is implemented through two reusable components: an inter-type declaration (ITD) component which is then extended by a pointcuts and advice component. This separation enables ITDs to be used independently of pointcuts and advice, *e.g.* to implement ITDs in the JastAdd compiler itself.

The Polyglot and JastAdd based implementations are quite similar at the component level. The main difference is that AspectJ is divided into two components in the latter solution. However, we see no reason why the Polyglot based frontend could not be split into the same structure. The difference in size between the two approaches is, however, quite striking: each JastAdd component is less then half the size of the corresponding Polyglot component. In Section 3 we analyse why this is the case, and also show additional benefits of the JastAdd based implementation in terms of improved localisation of concerns within each component.

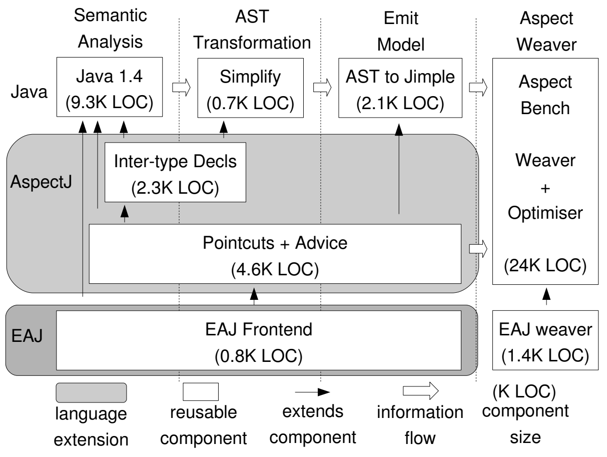

**Figure 2: Overview of the JastAdd based frontend in *abc*. Components are annotated with their sizes in lines of code. Java 1.4 is modularly extended to support AspectJ and EAJ.**

## 2.2 Polyglot extension mechanisms

Polyglot [23] is a frontend for Java intended for implementing extensions to the base language. By default, Java source code is parsed into an abstract syntax tree; then, Polyglot performs all the static checks required by the Java language in a number of passes which may rewrite the tree. The output is an annotated Java AST, complete with type information, which is written back to a Java source file that any standards-compliant Java compiler should be able to process. In the context of *abc*, Polyglot is used as a frontend for Soot, and so the *Java to Jimple* module inside Soot compiles the final AST into the Jimple intermediate representation, suppressing the output of Java files. It should be noted that the discussion in this paper pertains to Polyglot 1.3.*, the basis of *abc*.

Polyglot uses a number of design patterns to achieve extensibility while remaining in the realm of pure Java. Chief among them is an intricate system of *delegate and extension objects* that allows functionality to be inserted in the middle of the AST hierarchy. Each AST node has a corresponding concrete class and also an interface; moreover, it is associated with an extension and a delegate object which can be used to add extra information to existing nodes, or to modify the behaviour of particular methods, as long as the overriding conventions of always calling instance methods through the delegate are enforced.

In order to be able to change or extend existing AST nodes, another convention is necessary — such classes can only be instantiated through the *Factory* pattern. This means that rather than calling the constructor directly, a method on the AST node factory is used throughout the code that in turn allocates the object; if necessary, this method can be overridden in a subclass of the factory, allowing the programmer to replace all instances of one node with an extended version.

In addition to AST nodes, Polyglot has type nodes that represent the types and signatures of classes, methods, and fields. Like AST nodes, type nodes are constructed using a factory for extensibility. During compilation the AST is annotated with type nodes, once the names in the program have been resolved.

Finally, the transformations of the AST are implemented by strictly following the visitor pattern. A number of generic visitor passes are performed; each AST node includes generic methods for traversing its children, and can optionally rewrite nodes it visits to implement transformations and disambiguations. By performing appropriate actions at each node, such visitor passes are used to implement the static checks required by the Java specification (*e.g.* type checking, definite assignment *etc.*); in *abc*, they also implement some of the transformations required for AspectJ.

Note that each pass performs one specific task, and is meant to update the input tree non-destructively. As a result, it is easy to insert new passes in between existing ones to implement new functionality; indeed, *abc* augments Polyglot's default 16 passes to 42. It is worth pointing out that each pass does significantly less work than traditional compiler passes to allow for more fine-grained extensibility, so these large numbers aren't entirely surprising. It is up to the developer of an extension to determine where in this multitude of passes their own passes should be inserted; execution always follows the pre-determined order.

### 2.2.1 *Illustration: A Small Extension*

To make the discussion above a little more concrete, we will briefly discuss a small extension to the AspectJ language, focussing on Polyglot's extensibility mechanisms rather than *abc*'s.

We would like to implement the feature of *global pointcuts* (which actually form a part of the EAJ extension to AspectJ). This feature would allow us to specify a particular pointcut that restricts the applicability of every piece of advice in aspects matching a certain class-name pattern. We can write something like **global** : ∗ : !**within**(Hidden); to ensure that no advice could ever apply to the *Hidden* class, or **global** : Aspect : !**within**(Aspect); to prevent advice in a particular aspect from applying to itself. Thus, the general form of a global pointcut declaration is

**global** : <ClassNamePattern> : <Pointcut>;

The first step in implementing this as an extension to *abc* is to define the new keyword in the lexer and augment the parser specification with new productions describing the syntax. We will skip the details, as they aren't immediately relevant to Polyglot and are also covered elsewhere [4].

Then, we need to define a new AST node for the newly added language construct. As mentioned above, it is crucial to adhere to the rigorous use of factories and interfaces to create new nodes and invoke their members, respectively. Thus, we define a new interface for a global pointcut declaration, listing any public methods we might need in it:

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

The method will be called to insert the pointcut into a static data structure keeping track of all global pointcuts in the program. Of course this interface also offers all public members declared in its superinterface, PointcutDecl.

Then, we need to write a concrete class implementing the interface, which is given the name GlobalPointcutDecl_c by convention. Quite a bit of boilerplate code is required (a constructor and methods that allow visitors to traverse or rewrite the node generically); of course a concrete implementation for the method registerGlobalPointcut() should also be provided.

In order to instantiate this new class, we need to define a new AST node factory extending the existing version; this provides the following new method:

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

This method is invoked by the semantic actions in the extended parser to create appropriate AST nodes.

Finally, to actually implement the functionality of our new extension, we need to add some new Polyglot passes that know how to validate the global pointcut AST node and transform it. Concretely, we need two phases: First, all global pointcuts need to be collected, and then each pointcut in aspects matching the pattern must be conjoined with the pointcut of the corresponding global pointcut declaration. For reasons of brevity, one might combine the two passes into a single class called *GlobalPointcuts* and use a field to toggle between the two modes of operation. The following snippet illustrates the behaviour of the 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);
}
```

Thus, if we are entering a node of the appropriate type, we cause it to register itself with the static data structure for global pointcuts, and then delegate the actual work of continuing the traversal to the superclass. The second phase of the transformation takes place in the *leave()* method, which is called when a visitor leaves an AST node and has the option of rewriting the node. If pass == CONJOIN and the current node is a pointcut expression, we return the conjunction of the node with each matching global pointcut.

With this, the work required for a purely frontend-based extension is complete.

## 2.3 JastAdd modularisation mechanisms

JastAdd is a meta-compiler system that combines ideas from the aspect oriented and compiler construction communities. It has been used successfully to implement a complete Java 1.4 frontend with multiple backends, full Java 5 support through modular extensions [10], and additional modules for optional type systems (*e.g.* non-null types [9]). The key to its extensibility is a declarative extension to Java based on object-orientation, inter-type declarations, and attribute grammars with support for reference attributes, nonterminal attributes, and circular attributes. These features allow extensions to be specified modularly and then combined automatically by the tool.

Specification order is irrelevant in a declarative specification formalism, and the criterion for grouping a set of attributes and equations into modules is purely the promotion of reuse and understandability. We usually decompose compiler extensions at two levels. First we decompose the system into components based on large parts of functionality that are expected to be reused. One may for instance want to extend a Java 1.4 compiler with the following components: Java 5 (*i.e.* generics, attributes, enumerations and enhanced *for*-loops, which could all be components in their own right), pointcuts and advice, and inter-type declarations. At a more fine-grained level we decompose a component into modules primarily based on understandability.

The JastAdd formalism extends Java with two new kinds of modules: abstract grammar modules for specifying abstract syntax trees (ASTs), and attribute modules for specifying behaviour in a declarative fashion. These modules can be used from and combined with imperative Java classes.

### 2.3.1 Abstract grammars

JastAdd uses a small domain-specific language to define abstract grammars and generates Java classes that are used as AST nodes. The abstract grammar defines both an inheritance hierarchy and a composition hierarchy. Consider the code snippet below, which will be our running example for the rest of this section:

```
Program ::= Stmt;
abstract Stmt;
Block : Stmt ::= Stmt*;
EmptyStmt : Stmt;
```

Each production defines a node type and if applicable the kind of children that node may have. The example defines four node types: Program, Stmt, Block, and EmptyStmt. Block and EmptyStmt both inherit Stmt, which is abstract and used to hold behaviour common to all kinds of statements. Arbitrarily deep class hierarchies can be formed this way to include many abstraction levels. Composition is defined by adding an '::=' to a production and stating the desired children. The example states that a Program has a single child of type Stmt, while a Block has a list of children of type Stmt. The generated classes contains getters and setters used to traverse node children and build the AST respectively. Each node implicitly inherits the special class ASTNode, which provides a generic way to traverse the AST. More complex traversal patterns can be implemented by overriding a traversal in specific node classes. The following code snipped defines a traversal that visits the AST but stops at Block nodes:

```
public void ASTNode.visitStopAtBlock() {
  for(int i = 0; i < getNumChild())
    getChild(i).visitStopAtBlock();
}
public void Block.visitStopAtBlock() { }
```

### 2.3.2 Inter-type declarations

Inter-type declarations can be used to insert additional behaviour into an existing class hierarchy. JastAdd takes an extreme approach where all behaviour is defined using ITDs injected into the AST nodes generated from the abstract grammar. The ITDs supported in JastAdd differ slightly from AspectJ. Methods, constructors, and fields may be introduced in existing classes by explicitly naming the target class. Those members may also be introduced through interfaces, as long as there is no visible implementation in any class implementing the target interface. Declare parents differs from AspectJ in that only interfaces may be added to classes, and that those classes must be explicitly enumerated without using patterns.

Locally declared methods and inter-type methods may be replaced by another inter-type method declaration. This enables an extension to refine the behaviour in an existing component without editing the component source files. The method to be refined is explicitly named and if several refinements apply to that same method, they need to be explicitly ordered. The method being refined may be called from the refinement, if the original behaviour is need in computing the new behaviour. From an AspectJ viewpoint this corresponds to intercepting a method-execution join-point and applying around advice. This is the only pointcut and advice like construct supported in JastAdd and applies to methods, constructors, and attribute equations.

### 2.3.3 Declarative attributes

Compilation tasks are usually implemented in JastAdd using attribute grammars where attributes are defined by equations. The equations are defined in a syntax-directed style, solving a problem for each AST node in isolation. The order in which these equations are specified is irrelevant: the underlying attribute evaluation engine orders the evaluation of individual equations and combines them into a global solution. Attributes enable arbitrary decomposition which in turn promotes reuse and better localisation of concerns. Attribute grammars were first introduced by Knuth [20] and numereous extensions have been proposed over the years. Of particular interest for this work is the support for reference valued attributes [16], allowing computations on graph structures rather than trees, and cirular attributes [22], enabling automatic fixed-point iteration for mutually dependent attributes.

*Synthesised attributes* are quite similar to virtual methods without externally visible side-effects. An attribute is specified in a class and equations for that attribute may be overridden in subclasses. The requirement of no side-effects allows the results from such methods to be cached for efficiency. Consider the example below computing the total number of statements in a subtree. Each Stmt has a synthesised attribute called size(). Since there is no equation in Stmt this can be considered an abstract method with the additional constraint that overriding methods must be free from side-effects. The equation in Block iterates over its children, adding the sizes of subtrees. The size of the EmptyStmt is 1 and defined using a convenient functional style syntax which can be used when the value is a single expression. In our experience, the functional syntax promotes dividing a problem into smaller and smaller subproblems until the solution is almost trivial. The individual parts are then composed automatically by the underlying attribute evaluation engine.

```
// number of statements in subtree
syn int Stmt.size();
eq Block.size() {
  int size = 1; // a block is itself a statement
  for(int i = 0; i < getNumStmt(); i++)
    // add the contained statements
    size += getStmt(i).size();
  return size;
}
eq EmptyStmt.size() = 1;
```

*Inherited attributes* propagate information about the current context downwards in the tree and decouple the use of an attribute from its definition. The node reading an attribute need not be aware of which node defines that value but only that there is an ancestral node providing an equation. Consider the example below, which determines whether a statement is nested in another statement or not. The Stmt need not know if it is a child of a Program node or a Block node — only that there is an equation for the nestedInStmt() attribute. This is a remarkably simple way to describe properties dealing, for instance, with containment and visible declarations, i.e., it enables abstraction over the current context.

```
inh boolean Stmt.nestedInStmt();
eq Program.getStmt().nestedInStmt() = false;
eq Block.getStmt().nestedInStmt() = true;
```

### 2.3.4 The attributed AST

Synthesised and inherited attributes can easily be combined to compute new properties. Suppose, for example, that we would like to determine, for each statement, the outermost enclosing statement. We can achieve this by combining the inherited attribute enclosingStmt(), which gives the immediately enclosing block (if there is one), with the nestedInStmt() attribute defined above: For statements which are not nested, the outermost enclosing statement is just **this**; for all others, it is the outermost enclosing statement of the immediately enclosing statement.

```
inh Stmt Stmt.enclosingStmt();
eq Block.getStmt().enclosingStmt() = this;

syn Stmt Stmt.topStmt() =
     nestedInStmt() ? enclosingStmt().topStmt() : this;
```

Note that in the above, both enclosingStmt() and topStmt() are so-called *reference attributes* — attributes that hold references to other AST nodes. Since each compilation task has been recast into the problem of defining attributes on the AST, such reference attributes turn out to be extremely useful. Consider, for example, name analysis: Each name used in a program should be bound to the corresponding declaration, according to the scoping rules of the language. We can therefore define an attribute on each name node whose value references the declaration.

In general, reference attributes allow us to superimpose graphs onto the AST, which can then be used by other attributes defining additional analyses. This makes it straightforward to use the AST as the only data structure, eliminating the need for external data structures like symbol tables. The main benefit is that now the same extension mechanisms that apply to ASTs and attributes can be used for all compile-time data structures. A more thorough description of how to combine attributes (including circular attributes) to implement extensible name binding, type analysis and definite assignment checking for Java can be found in [8–10, 22].

#### *2.3.5 Illustration: A small example revisited*

For completeness, we will briefly indicate what steps would be necessary to implement the purely frontend-based extension of global pointcuts discussed in Section 2.2.1 in JastAdd. Again, we will assume that the parser and lexer have already been extended so that the syntax of our extension is recognised and appropriate ASTs are built.

Rather refreshingly, there is no boilerplate code to write, so we can jump straight into the semantic work that was done by two AST visitor passes in Polyglot. First of all, we need to collect all global pointcuts from the entire program into a single list. This can be achieved with the following attribute definitions:

```
inh lazy List BodyDecl.globalPointcutDecls();
eq Program.getCompilationUnit().globalPointcutDecls() =
    collectGlobals(new List());

syn ASTNode.collectGlobals(List globals) {
    for(int i = 0; i < getNumChild(); i++)
        globals = getChild(i).collectGlobals(globals);
    return globals;
}
eq GlobalPointcutDecl.collectGlobals(List globals) =
    globals.add(this);
```

We use a combination of inherited and synthesised attributes: collectGlobals() traverses all children of the current node and adds global pointcuts to a list, which is exposed to all AST nodes as the inherited attribute globalPointcutDecls(). Now it is trivial to modify the code generation for each PointcutExpr node to conjoin the global pointcuts with itself.

| AST NODES | VISITORS |
|---|---|
| Patterns (30) | Pattern matching (5) |
| Pointcut designators (25) | Pointcut dependency analysis (1) |
| Advice and named pointcut declarations (20) | Inter-type declarations (3) |
| Inter-type declarations (5) | Name mangling (2) |
| Declare parents (6) | Declare parents (2) |
| EXTENSION NODES | Initialising backend (6) |
| Inter-type declarations (14) | Aspect precedence (2) |
| TYPES | Building new types (1) |
| Pointcuts, aspects, and advice type-instances (3) | Miscellaneous checks (2) |
| Inter-type declarations (3) | |
| Aspect-aware context (2) | |

**Figure 3: Decomposition of the Polyglot-based *abc* frontend. The parenthesised numbers show the number of files for each concern.**

| INTER-TYPE DECLARATIONS | PATTERNS, POINTCUTS, AND ADVICE |
|---|---|
| Declare parents (2) | AspectJ AST (1) |
| Declare precedence (2) | Advice (1) |
| Inter-type common code (2) | Implicit methods (1) |
| | Implicit variables (1) |
| *(the following have a file each for: the AST, name-analysis, error-checking, and code generation)* | Declare error / soft (1) |
| | Patterns (1) |
| | Pointcuts (1) |
| Inter-type fields (4) | TRANSLATION TO BACKEND STRUCTURES, AND CODE GENERATION |
| Inter-type methods (4) | Advice (1) |
| Inter-type constructors (4) | Class and method categories (1) |
| INTERACTION BETWEEN MODULES | Patterns (1) |
| Declare precedence / backend structures (1) | Pointcuts (1) |
| ITDs / code generation (1) | |

**Figure 4: Decomposition of the JastAdd-based *abc* frontend. The parenthesised numbers show the number of files for each concern.**

## 3. CASE STUDY: EXTENSIBLE ASPECTJ FRONTENDS COMPARISON

### 3.1 Inter-type declarations

The pervasive use of inter-type declarations in the JastAdd-based frontend results in a starkly different modularisation to that of the Polyglot-based frontend, as shown by Figure 3 and Figure 4. There is less scattering of related code in the JastAdd-based frontend, and the AspectJ language concepts and constructs map, on the whole, directly to the module structure.

As discussed in Section 2.2, there are broadly three kinds of classes in Polyglot—AST nodes, types nodes, and AST visitors. This leads to the related code being split into separate files. For example, note the division of the code for inter-type declarations into 11 files in 3 directories. Although dividing code into smaller, simpler modules is the programmer's main weapon against complexity, the divisions in Figure 3 are not information-hiding: they therefore hinder, rather than help, understanding.

In contrast, the freedom of using ITDs to declare attributes on any node, from any implementation file, allows the code to be carefully structured around the concepts of the problem domain — in this case, the language constructs of AspectJ. In particular, note that there are modules listed in Figure 4 that do not appear at all in Figure 3. One such module is *Implicit Variables*, which implements support for variables representing the current joinpoint — *thisJoinPoint*, *thisJoinPointStaticPart*, and *thisEnclosingJoinPointStaticPart*. This module does not appear in the modularisation of the Polyglot-based frontend because its implementation is split up into little pieces of code in the *local*, *advice*, and *if-pointcut* AST classes, and the visitor for propagating data needed for the compiler to generate methods.

Another effect of using ITDs is that the need to use factories for instantiating AST and type objects is eliminated.

Polyglot uses factories so that any classes of the base-compiler can be replaced with classes implementing new or different behaviour. For maximal flexibility, the Polyglot convention is to have a separate interface and implementation class for every AST and type node. By using ITDs, the code that would have been put into a new class can simply be injected into the old. The *Factories/interfaces* line of Table 1 shows the resulting savings in lines-of-code[1]. This additional code was time-consuming and error-prone to write, and arguably obscured the intention of the rest of the code, so the use of ITDs was a significant improvement.

| | abc (Polyglot) | | abc (JastAdd) | |
|---|---|---|---|---|
| | lines | files | lines | files |
| Factories/interfaces | 5104 | 102 | 0 | 0 |
| Nodes and members | 436 | 109 | 94 | 7 |
| Complete frontend | 16745 | 254 | 4582 | 41 |

**Table 1: The overheads of factories and class definitions, compared to that of ITDs and a DSL for generating AST nodes**

Table 1 also demonstrates the effect of using JastAdd's domain-specific language for defining the inheritance hierarchy and structure of AST nodes. For the Polyglot-based frontend, the number of lines devoted to *Nodes and members* was calculated by counting only those lines of AST source files that are part of the class definition itself (i.e. name/interfaces/superclass) or declare fields for child AST nodes. Grammars are commonly used to concisely and clearly explain the structure of languages and data-structures, and the JastAdd-based frontend benefits much more from this clarity than it does from the reduced number of lines-of-code: one complaint the authors have often experienced and heard from colleagues is that they found it very difficult to initially understand the relationship between the many different AST nodes in Polyglot and the Polyglot-based *abc* frontend.

## 3.2 Phases and Demand-Driven Evaluation

The original *abc* frontend performs 42 explicitly-scheduled passes over the program AST, using visitors from Polyglot, JavaToJimple, and the Polyglot-based AspectJ frontend itself. The main reason for the large number of passes is that most non-trivial transformations or computations performed on the AST require more than one pass.

The global-pointcut-declaration example described in Section 2.2.1 is illustrates a two-pass visitor: before one visitor could transform the advice-pointcuts in the program according to the global-pointcut declarations, those declarations had to first be collected by a separate pass. If the declarations were collected by the same tree-walk as the transforming pass, then global-pointcut declarations would only apply to advice that was textually after them! The transformation pass has a data-dependence on the collection pass, and therefore also a temporal dependence.

The trouble with the large number of passes is that the temporal dependencies between them are implicit and brittle. Whilst implementing the global-pointcut-declarations extension, it took more than an hour of experimentation to find the right order to run the new passes in relation to the old.

The JastAdd-based frontend has only 5 explicitly-scheduled passes: 3 for checking or transforming the AST (reporting errors, weaving inter-type declarations, and flattening nested classes), and two for code generation. Although the frontend contains other code for (partial) tree-walks and more complicated traversals, they are all scheduled dynamically and automatically. This is possible because attributes are evaluated on-demand, with optional caching for attributes which are expensive to compute.

For example, recall that in the JastAdd code for implementing global-pointcut-declarations shown in Section 2.3.5, the list of all declarations is returned by the inherited attribute globalPointcutDecls(). This attribute may be evaluated at any time — automatically triggering the traversal to collect all the relevant declarations. Since the attribute is marked **lazy**, its value is cached after being evaluated once.

Demand-driven evaluation has an important effect on modularity: code can be decomposed by activity, rather than according to temporal constraints. The resulting modules are also more easily composed with new modules because there is no need to manually discover and track temporal dependencies, manually merge long lists of passes, or even break what would have been circular temporal dependencies (unless, of course, the actual data-dependencies are also circular).

Even industrial-strength compilers can suffer from unexpected surprises due to the combination of global data structures and mis-scheduling. Consider the AspectJ program below where the inheritance hierarchy is changed using declare parents and the class Middle is inserted in between Lower and Upper. The Lower class has a single field of type Inner. That type is only visible if the declare parents clause is considered when looking up member types in Lower. The current version of *ajc* gives a strange result when compiling this program. If the member class Unrelated is included in Upper then Inner is bound correctly, but if we remove Unrelated then Inner can not be found. The unrelated member type in Upper will thus cause member classes made visible through declare parents to be considered. These members are, however, not considered otherwise. In our declarative implementation of member type lookup we can change a single equation to either include declare parents, the approach taken in *abc*, or exclude declare parents, the intended behaviour in *ajc*, when computing member types.

```
class Upper {
  // if we remove Unrelated then ajc
  // fails to bind Inner in Lower
  class Unrelated { }
}
class Middle extends Upper {
  class Inner { }
}
class Lower extends Upper {
  Inner inner;
}
aspect Aspect {
  declare parents: Lower extends Middle;
}
```

## 3.3 Traversal and Inherited Attributes

Recall from Section 2.3.4 that inherited attributes provide an AST node with simple and flexible access to information

[1]All line counts were generated using SLOCCount by David A. Wheeler

about its environment or context. Whilst a synthesised attribute is evaluated by traversing the AST top-down, inherited attributes are implemented in the JastAdd system by traversing *up* the tree. In contrast, Polyglot visitors only traverse the tree top-down and left-to-right. This section examines the consequences of this difference.

Polyglot visitors are well-suited to some problems. For example, the problem of resolving local variable names in well-nested blocks is straightforwardly solved in Polyglot by the visitor maintaining a stack-based context: upon entry to a block, a new scope is pushed onto the stack; the scope is populated by local variable declarations; the whole context is used to resolve names; and upon exit the top scope is popped off.

However, the problem of resolving types, methods and fields — in which uses may appear before definitions, and the definitions may be in a super-type of the current class — is less straightforward. It is solved in Polyglot by using monolithic data-structures, which are populated gradually during several separate passes over the whole AST: there are several kinds of name-disambiguation passes, which are followed by a type-building pass.

Section 2.3.4 outlined how the same problem is solved using JastAdd: combinations of inherited and synthesised reference attributes, when evaluated and cached, form a graph superimposed on the AST — the AST is the sole data-structure. This has several advantages. Firstly, since the attributes are evaluated on-demand, time and memory is conserved in comparison to populating large monolithic data-structures with data when much of it may not be used (execution time measurements are shown in Section 3.4). Secondly, the algorithms used are more easily understood because they are not broken up into parts residing in separate visitor classes. Finally, the attributes are much more flexible than monolithic data-structures. In the Polyglot-based *abc* frontend, the pattern-matcher constructs a completely separate representation of the program's package and class structure than that used by the type-checker — despite much of the data being equivalent — because the monolithic data structures used by the latter were not quite flexible enough. In contrast, implementing the pattern matcher in the JastAdd-based frontend was greatly simplified by using the pre-existing attributes for navigating the inheritance hierarchy of the program.

### 3.3.1 Inherited attributes using AspectJ

Fans of Laddad's AspectJ textbook, *AspectJ in Action*, may be thinking at this point that an inherited attribute sounds similar to the *wormhole* pattern described in the book. Indeed, both involve capturing context and using it without explicitly passing it through any intermediate stages. Note, though, that when referring to inherited attributes, context means the nodes above an AST node on the path to the root; when referring to the wormhole pattern, context refers to the methods on the call-stack. Since these two notions of context correspond for a standard recursive implementation of a tree-walk, an AspectJ equivalent is shown below for the *globalPointcutDecls()* inherited attribute described in Section 2.3.5, along with the original JastAdd version.

Firstly, using AspectJ and the wormhole pattern (together with caching):

```
aspect CollectGlobalPointcutDecls
{
    pointcut context(Program prog):
        call(void Program.*(..)) && target(prog);

    pointcut globalPointcutDecls():
        call(List BodyDecl.globalPointcutDecls());

    public List BodyDecl.globalPointcutDecls()
    {
        throw new RuntimeException(
            "should␣always␣be␣intercepted!");
    }

    private List BodyDecl.cachedGPDs = null;

    List around(Program prog):
        cflow(context(prog)) && globalPointcutDecls()
    {
        if (prog.cachedGPDs == null)
            prog.cachedGPDs =
                prog.collectGlobals(new List());
        return prog.cachedGPDs;
    }
}
```

Secondly, the JastAdd equivalent:

```
inh lazy BodyDecl.globalPointcutDecls();
eq Program.getCompilationUnit().globalPointcutDecls() =
    collectGlobals(new List());
```

## 3.4 Computational Cost

A natural question that arises is, of course, what price must be paid in terms of performance. Demand-driven evaluation has some overheads compared to a hand-optimised solution written purely for speed; also, a certain amount of generality is unavoidable when trying to design an extensible framework, and this in turn adds to execution time.

In this section, we shall demonstrate that the runtime costs of employing the declarative JastAdd system are not at all prohibitive — in fact, performance is significantly better for the JastAdd-based frontend (*abc-ja* for brevity in what follows) than for the original *abc*, written in pure Java and using Polyglot's extensibility mechanisms. An in-depth study of performance considerations is beyond the scope of this paper, however, and the interested reader is referred to [10] for a more thorough treatment.

To test the performance of the AspectJ extension, we compiled AJHotDraw [28], an aspect-oriented refactoring of JHotDraw [11] that makes heavy use of ITDs, pointcuts and advice, *declare parents*, *declare soft* and other features of AspectJ. Also, since Java is (mostly) a subset of AspectJ, we repeated our measurements with the original pure-Java version of JHotDraw.

Our results are summarised in Table 2. Generally, compilation time with *abc* is dominated by the backend, Soot, which performs some expensive analyses and optimisations. This is the reason why we give a separate running time for just the frontend passes of the compiler, which is, of course, the only difference between *abc* and *abc-ja*. We can see that typically the time spent in the front-end is reduced by more than 50%, and correspondingly the overall runtime drops by around 25% with the new frontend.

These observations are roughly in line with the previously reported performance numbers for the JastAdd system as a Java compiler, and so should come as no surprise. The fact that we can achieve such better performance with a smaller amount of better structured code is particularly pleasing.

| Benchmark | Lines of Code | *abc* (frontend) | *abc* (full) | *abc-ja* (frontend) | *abc* (full) |
|---|---|---|---|---|---|
| AJHotDraw | 21055 | 44.35s | 84.41s | 21.80s | 64.73s |
| JHotDraw | 28385 | 51.68s | 97.42s | 24.21s | 72.83s |

**Table 2: Times for frontend passes and full compilation for our benchmarks**

## 4. DISCUSSION

In Section 3 we compared a pure object-oriented implementation in Java with an aspect-oriented solution implemented using JastAdd. In this section we discuss the main differences in implementing an AspectJ frontend from an extensibility and modularity point of view. We also try to generalise some of our conclusions to either the domain of extensible compiler construction or aspect-oriented software development at large.

The classical decomposition of a compiler is into an underlying tree structure and a number of computations operating on top of that structure. Such computations are often implemented using the visitor pattern in object-oriented languages. Extensibility of such structures is traditionally a problem and many compiler frameworks support extensions to either the tree structure or to the computations, but not both. This particular problem has been called the *Expression Problem* in recent years, and there have been many proposals of extensible visitors, extended type systems, and new language features to solve the problem.

Polyglot is based on an extensible visitor pattern which allows for such decomposition and extension in both dimensions. Combining modules with different decomposition criteria is, however, cumbersome. Consider the scenario where a base compiler is decomposed in a tree structure and a set of analyses, while extensions are decomposed in one module for each new language construct. Each language construct needs to extend one or more visitors and these new visitors need to be explicitly combined either through a linear inheritance hierarchy or a visitor pattern supporting composition of visitors. Both solutions introduce unnecessary dependencies between unrelated language constructs, which makes it harder to combine extensions.

Inter-type declarations provide a more flexible solution in which the addition of node types and new methods in existing node types can be modularised freely, particularly when combined with declarative attributes, as discussed below. Visitor patterns also rely on some glue code being written to enable extensibility. This is not only tedious but also error prone, which we ourselves experienced while using Polyglot: If the pattern is not followed correctly then the solution is not extensible. A possible drawback of using inter-type declarations is that a traversal pattern defined using introduced methods can not be reused in the same way as a visitor. This is not, in our experience, a problem since traversal patterns are either very simple, e.g., a bottom-up-left-right traversal, or so complex that they are not reused. We implement simple traversals using the generic traversal functionality provided in the top node in the JastAdd AST hierarchy. If a complex traversal needs to be reused it can be parameterised by an argument containing the actual computation for each node. Another drawback is that most systems supporting ITDs require a full program analysis during compilation.

An often overlooked problem is how handling of dependencies between different analyses affects modularity. Consider the problem of name resolution and type binding in an object-oriented language. The type of an expression accessing a field depends on name resolution, since the field name needs to be looked up and associated with a corresponding declaration to determine its type. However, it is perfectly fine for such a field to be defined in a superclass or even an implemented interface. We thus have to compute the inheritance hierarchy prior to looking up fields, which makes name resolution depend on type binding. Therefore, name resolution and type binding are mutually dependent.

A visitor based solution would solve this problem by scheduling a number of traversals over the AST where passes share information through external data-structures, *e.g.* symbol tables. Such structures are usually hard to extend modularly to support new language features. Moreover, when adding inter-type declarations to a language we introduce new dependences between name resolution and type binding and must therefore schedule new passes which affect the information in the symbol table. Incorrect scheduling may result in strange behaviour such as the example with inner classes in Section 3.2. Dependencies between computations thus restrict the way we can modularise visitors and require error prone scheduling of visitor passes.

These problems are alleviated by using declarative attributes in JastAdd which provides two new modularisation mechanisms: demand-driven evaluation and inherited attributes.

This enables arbitrary modularisation at the attribute level, which makes it trivial to decompose the system according to any desired criterion.

The functional programming community has for a long time acknowledged lazy evaluation as an important modularisation mechanism [18]. The demand-driven evaluation used in JastAdd is a simplified form of lazy evaluation where attributes are evaluated lazily but arguments are evaluated strictly. It allows us to specify general properties for AST nodes at various levels of abstraction and only pay the evaluation cost if they are actually needed.

Another advantage of the declarative model with fine-grained attributes is that, in our experience, it suffices to consider only a very simple join-point model. JastAdd only supports method execution interception with only one type of advice: around advice. Moreover, the pointcut language only supports explicit naming of a single method with a fully qualified name. There is thus no support for quantification or method patterns, and no dynamic pointcuts. We believe that part of the reason that we do not need quantification is that we operate on an AST with a deep class hierarchy. There is often a common superclass that can host the behaviour shared by many node types. In a few cases we added a new interface, used declare parents to make some classes implement that interface, and introduced methods with shared behaviour through that interface. This is admittedly a form of quantification, but in a very controlled fashion using only explicit enumeration. Another reason may be that in a declarative model it does not really make sense to point out a series of execution points. We conclude that, in our case-study, we not have benefited from a richer pointcut language or more dynamic aspect features.

## 5. RELATED WORK

The structure of *abc* as well as extensible compiler construction using JastAdd have been presented elsewhere by one or more of the authors [1,10]. In this paper we build on that work and give a detailed comparison between a state-of-the-art object-oriented solution and an aspect-oriented solution. We also present an improved platform for AspectJ language research and discuss how the requirements on modularisation mechanisms can be met using a combination of aspect-oriented and compiler construction techniques.

Extensible computations on tree structures are often implemented using an extensible visitor pattern, e.g., Polyglot [23] or Functional Visitors [7]. AspectJ can be used to improve the pattern as shown in [13]. Adaptive programming improves on visitors by making extensible computations structure shy, in particular when used with Demeter Interfaces [26]. These approaches all enable flexible extension of individual phases, and even composition of phases, but they still rely on manual scheduling for ordering different computations. In our pure object-oriented AspectJ frontend we were required to schedule up to 45 difference passes manually, which turned out to be both cumbersome and error prone.

Static features of AOP are commonly used in compiler construction courses, *e.g.* [17,25], and even with course projects implementing subsets of AspectJ [19]. We are not aware of any courses that focus on extensible compilers or combine modularisation mechanisms from the aspect-oriented and compiler construction communities.

There are several compiler construction tools that incorporate ideas from the aspect oriented community. Rebernak *et al.* give a brief survey of such tools in [24]. We are not aware of any other large-scale experiment that evaluates benefits from using AOP compared to a pure object-oriented approach. Hausl implemented a subset of AspectJ for the Steamloom Virtual Machine [15]. That work is based on JastAdd as well but does not evaluate extensibility of modularisation mechanisms.

This paper is focused on the frontend of a compiler dealing with static semantic analysis, but there are other approaches to extensibility dealing with different phases of the compilation process. Bravenboer *et al.* showed how parsing ambiguities in the context of AspectJ can be handled in an elegant fashion using Generalised LR parsing [6]. Kojarski and Lorenz present the composition framework AWESOME, which is a multi-language aspect weaver [21]. Bockisch and Mezini created a meta model for pointcuts and advice which allowed them to completely separate the frontend from the backend in their AspectJ compiler [5]. We believe that our frontend could easily be extended to instantitate that meta model and then benefit from alternative backends. A similar model, albeit unpublished, is available in *abc* having a Polyglot and a JastAdd based frontend sharing a common interface to the weaver and optimiser. The described techniques and frameworks are complementary to the research presented in this paper and it would be interesting to combine them to enhance extensibility in all phases of an AspectJ compiler.

## 6. CONCLUSIONS

We have carried out a large scale experiment in modular compiler construction using the Java based framework Polyglot and the aspect-oriented compiler-construction tool JastAdd to implement an extensible AspectJ frontend. Both solutions are integrated in *abc* and reuse the same backend for optimisation and aspect weaving.

By combining AOP with modularisation techniques from the compiler construction community we gained a more flexible frontend where extension components are composable and concerns better localised. Components can be modularised freely using different criteria, e.g., according to computation task or as new language constructs. Moreover, the JastAdd based implementation is about half the size of the Polyglot solution, while still being twice as fast for AspectJ programs as well as normal Java programs.

A substantial part of the difference in size is accounted for by using inter-type declarations instead of an extensible visitor pattern, thereby eliminating a lot of boilerplate code such as factories. ITDs also result in less scattering of related code, and AspectJ language concepts and constructs map directly to the module structure. There are, however, often dependencies between various phases in a compiler frontend and such computations need to be scheduled manually when using imperative tree traversals. State shared between passes need to be stored in auxiliary data structures passed from one traversal to the other. Mutually dependent analyses, *e.g.* name binding and type analysis for object-oriented languages, need to be carried out in multiple passes, often resulting in a single concern, like name binding, being scattered over multiple modules. The Polyglot solution contains 45 traversals that need to be scheduled manually. The JastAdd solution, on the other hand, contains only a few phases — such as static semantic analysis, AST simplifications, and backend model instantiation — while keeping concerns localised.

Demand-driven evaluation with caching and attribute grammars provides just the right tools for describing such analyses in a declarative way where the scheduling is carried out automatically by an underlying attribute evaluation engine. This has an important effect on modularity: code can be decomposed by activity, rather than according to temporal constraints. Demand-driven evaluation has also proven efficient in practice, as our JastAdd based frontend outperforms the Polyglot based solution by a factor of two. We believe that the AOP community at large can benefit from viewing demand-driven evaluation as an important modularisation mechanism. Reference attribute grammars generalise attribute grammars to graphs rather than trees and can enhance extensible implementation of graph-based computations relying on context-sensitive information.

In the near future we plan to implement AspectJ 5 by combining the AspectJ extension with a JastAdd based Java 5 extension. It will be particularly interesting to see how these extensions interact and how to express the integration points. In the longer term, we intend to experiment with the same modularisation mechanisms to improve backend extensibility in *abc*.

## 7. REFERENCES

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien

Sereni, Ganesh Sittampalam, and Julian Tibble. The AspectBench Compiler for AspectJ. In *Generative Programming and Component Engineering*, volume 3676 of *LNCS*, pages 10–16. Springer, 2005.

[2] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications*, pages 345–364. ACM Press, 2005.

[3] Tomoyuki Aotani and Hidehiko Masuhara. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *AOSD*, pages 161–172, 2007.

[4] 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 *Aspect-Oriented Software Development (AOSD)*, pages 87–98. ACM Press, 2005.

[5] Christoph Bockisch and Mira Mezini. A flexible architecture for pointcut-advice language implementations. In *VMIL '07: Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, page 1, New York, NY, USA, 2007. ACM Press.

[6] Martin Bravenboer, Éric Tanter, and Eelco Visser. Declarative, formal, and extensible syntax definition for aspectj. In *Proceedings of OOPSLA '06*, pages 209–228, New York, NY, USA, 2006. ACM Press.

[7] Bryan Chadwick, Therapon Skotiniotis, and Karl Lieberherr. Functional Visitors Revisited. Technical Report NU-CCIS-06-03, Northeastern University, Boston, May 2006.

[8] Torbjörn Ekman and Görel Hedin. Modular name analysis for Java using JastAdd. In *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005*, volume 4143 of *LNCS*. Springer, 2006.

[9] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of non-null types for Java. *Proceedings of TOOLS Europe 2007, Journal of Object Technology*, 6(7), 2007.

[10] Torbjörn Ekman and Görel Hedin. The JastAdd Extensible Java Compiler. In David Bacon, editor, *Proceedings of OOPSLA 2007*, 2007.

[11] Erich Gamma. JHotDraw. Available from `http://sourceforge.net/projects/jhotdraw`, 2004.

[12] Robert Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–51, New York, NY, USA, 2006. ACM Press.

[13] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA*, pages 161–173, 2002.

[14] Bruno Harbulot and John R. Gurd. A join point for loops in AspectJ. In *AOSD*, pages 63–74, 2006.

[15] Michael Hausl. An AspectJ Compiler for the Steamloom Virtual Machine. `http://www.st.informatik.tu-darmstadt.de/pages/projects/ALIA/Frontend/ALIA_AJ_Compiler.pdf`, 2007.

[16] Görel Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), pages 301–317, 2000.

[17] Görel Hedin. Compiler Construction at Lund University, Sweden. http://www.cs.lth.se/EDA180/2006-web, 2006.

[18] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[19] Gregor Kiczales. Introduction to Compiler Construction at University of British Columbia, Canada. `http://http://www.cs.ubc.ca/~gregor/teaching/cpsc411/index.html`, 2006.

[20] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[21] Sergei Kojarski and David H. Lorenz. AWESOME: A Co-Weaving System for Multiple Aspect-Oriented Extensions. In David Bacon, editor, *Proceedings of OOPSLA 2007*, 2007.

[22] E. Magnusson and G. Hedin. Circular Reference Attributed Grammars - Their Evaluation and Applications. *Electr. Notes Theor. Comput. Sci.*, 82(3), 2003.

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

[24] Damijan Rebernak, Marjan Mernik, Hui Wu, and Jeff Gray. Domain-Specific Aspect Languages for Modularizing Crosscutting Concerns in Grammars. In *"Proceedings of Domain-Specific Aspect Languages Workshop at GPCE 2006"*, 2006.

[25] Michael Schwartzbach. Compilation Course at Aarhus University, Denmark. `http://www.brics.dk/~mis/dOvs`, 2007.

[26] Therapon Skotiniotis, Jeffrey Palm, and Karl J. Lieberherr. Demeter interfaces: Adaptive programming without surprises. In Dave Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 477–500. Springer, 2006.

[27] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[28] Arie van Deursen, Leon Moonen, and Marius Marin. AJHotDraw. `http://sourceforge.net/projects/ajhotdraw/`, 2006.

[29] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[30] Phil Wadler. The expression problem. Posted on the Java Genericity mailing list, 1998.