# The JastAdd Extensible Java Compiler

Torbjörn Ekman
Programming Tools Group
University of Oxford, UK
torbjorn@comlab.ox.ac.uk

Görel Hedin
Department of Computer Science
Lund University, Sweden
gorel@cs.lth.se

## Abstract

The JastAdd Extensible Java Compiler is a high quality Java compiler that is easy to extend in order to build static analysis tools for Java, and to extend Java with new language constructs. It is built modularly, with a Java 1.4 compiler that is extended to a Java 5 compiler. Example applications that are built as extensions include an alternative backend that generates Jimple, an extension of Java with AspectJ constructs, and the implementation of a pluggable type system for non-null checking and inference.

The system is implemented using JastAdd, a declarative Java-like language. We describe the compiler architecture, the major design ideas for building and extending the compiler, in particular, for dealing with complex extensions that affect name and type analysis. Our extensible compiler compares very favorably concerning quality, speed and size with other extensible Java compiler frameworks. It also compares favorably in quality and size compared with traditional non-extensible Java compilers, and it runs within a factor of three compared to javac.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors

***General Terms*** Design, Languages

***Keywords*** OOP, Compilers, Extensibility, Declarative Frameworks, Modularity, Java

## 1. Introduction

This paper presents the JastAdd Extensible Java Compiler, *JastAddJ*. It is built using the metacompiler tool JastAdd which provides advanced support for constructing modular and extensible compilers [HM03, Ekm06]. When using *JastAddJ* as is, it works in the same way as a normal Java compiler, generating class files from source code. What is new about *JastAddJ* is the way it can easily be extended with language constructs as well as with static analyses. As an example of language extension, *JastAddJ* itself is built as a base part supporting Java 1.4, and an extension that adds the new features in Java 5: enums, the enhanced for statement, autoboxing, varargs, static imports, and generics with wildcards. As an example of static analysis, modules for non-null type checking and type inference have been implemented as extension modules to *JastAddJ* [EH07]. Both of these examples illustrate highly non-trivial extensions.

The key to the extensibility of *JastAddJ* is the declarative foundation of the underlying system JastAdd, which is based on object-orientation, inter-type declarations [KHH$^+$01], declarative rewrites [EH04], and attribute grammars, including support for reference attributes [Hed00], nonterminal attributes [VSK89] and circular attributes [MH03]. These declarative features allow extensions to be built without interacting unduly with each other, and thereby making them easy to combine.

Writing a compiler for a complex language such as Java is notoriously hard and there are numerous subtle details that all need to be handled correctly to support the full language. In contrast to the other extensible Java compilers that we know of, *JastAddJ* is highly compliant with the language specification, actually passing a slightly higher number of tests in the Jacks test suite [jac07a] than popular compilers like javac and the Eclipse compiler.

To be of practical use, an extensible compiler needs to be able to handle large programs and also to be reasonably fast. *JastAddJ* can handle large programs, more than 100K LOCs, and it runs well within a factor of three as compared to javac. This is reasonably fast considering the main application area: to do experiments in language design, to add pluggable type systems [Bra04], and to build specialized analysis or transformation tools. *JastAddJ* is open source [jas07] and we have used it extensively in our own research projects.

The rest of this paper is structured as follows. In section 2 we discuss the main architecture of *JastAddJ*, the modules it consists of, and examples of how these modules have been reused and combined with extension modules in different projects. Section 3 discusses the major design ideas we have developed in building *JastAddJ*. Sections 4-7 discuss key as-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*OOPSLA'07,* October 21–25, 2007, Montréal, Québec, Canada.
Copyright © 2007 ACM 978-1-59593-786-5/07/0010…$5.00

pects of compilation and how they are handled in *JastAddJ* to support extensibility: name analysis, type analysis, and definite assignment. Section 8 evaluates *JastAddJ*, providing performance measures, etc., and comparing to other compilers or similar tools (both extensible and non-extensible). Finally, section 9 discusses related work and section 10 concludes the paper.

## 2. Architecture

### 2.1 Main components

*JastAddJ* consists of four main components: a *Java 1.4 frontend* and *backend*, and a *Java 5 frontend* and *backend*. Each component is represented as a directory of reusable source files (JastAdd files and parser generator input files), a main program (in Java), and a build file. The backends are *extensions* of the frontends: they reuse source files from the frontend components. Similarly, the Java 5 components are extensions of the Java 1.4 components, defining only what is needed to extend the Java 1.4 implementation to Java 5. New extended languages, analyzers or backends can be built by defining components that similarly extend the existing *JastAddJ* components.

The main programs in the frontends are error checking prettyprinters: they parse Java source files, read dependent class files, print compile-time error messages and print normalized versions of the files. These tools serve as examples for how to build source-to-source translating tools and analysis tools such as pluggable type checkers. The main programs in the backends are normal Java compilers that parse Java files, print compile-time error messages and produce class files.

Figure 1 shows the main components in *JastAddJ*. The numbers represent the total number of lines of source code (LOC) in the component measured using using SLOCCount [Whe07].

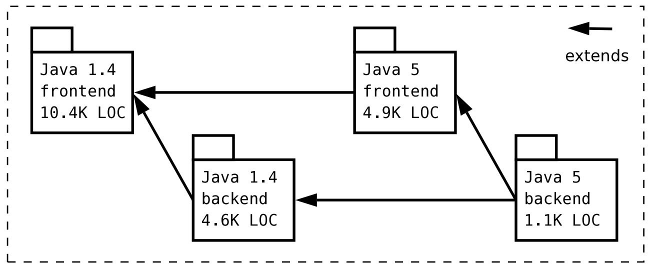

**Figure 1.** Main components of *JastAddJ*

### 2.2 Example extensions

We have made several different extensions to *JastAddJ*. One example is a pluggable type system for checking and inferring non-null annotations [EH07]. The main implementation of the non-null checker is built as an extension to the Java 1.4 frontend. It can be run as a pure checker (without compiling to bytecode) or as an extended compiler, combining it with the Java 1.4 backend. The extension can furthermore be combined with the Java 5 frontend and backend. Only a few attributes need to be refined when combining the non-null type system extension with the generic type system extension. Figure 2 depicts the different components and how they extend each other.

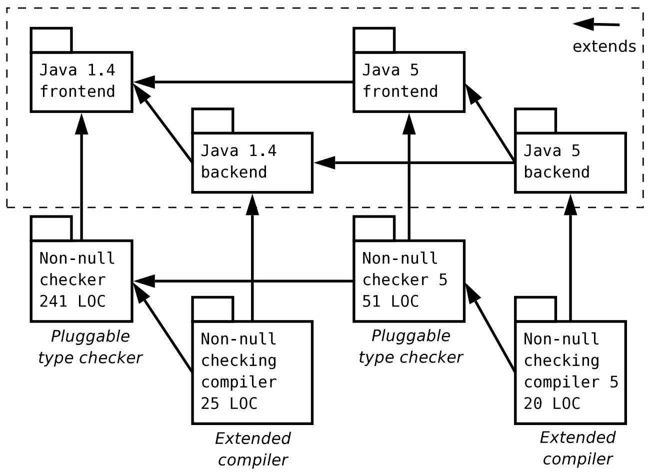

**Figure 2.** A pluggable non-null checker for *JastAddJ*

We are currently collaborating with the AspectBench compiler team in implementing an AspectJ frontend as an extension to the Java 1.4 frontend. As part of that work we have also implemented a backend that generates Jimple code for the Soot bytecode manipulation framework [VRHS$^+$99]. These two components are replacing the polyglot-based frontend of the abc AspectJ compiler [ACH$^+$06] by *JastAddJ*. Advice weaving is performed on the Jimple representation while inter-type declarations are woven in the *JastAddJ*-based frontend for AspectJ. A major benefit from this change is the automatic scheduling of attribute computations rather than relying on manual scheduling of more than 45 different passes in the polyglot-based frontend.

Other extensions include an alternative backend for PalVM, a Smalltalk-based virtual machine intended for low memory pervasive systems, developed in the EU integrated project PalCom [Pal07]; a Java to C compiler for hard real-time systems [Nil06], and a system for automatic hardware compilation for Java [And05].

### 2.3 Generation architecture

A *JastAddJ* component is specified by four principal parts: an *abstract grammar* defining the structure of abstract syntax trees (AST), *behavior specifications* defining the behavior of the AST, a *context-free grammar*, defining how text is parsed into ASTs, and a *main program* that reads the input file, runs the parser to build the AST, and uses the AST behavior to generate output.

The abstract grammar and the behavior are specified using the JastAdd language. The JastAdd tool takes these specifications and generates an object-oriented class hierarchy in

Java for the AST. For parsing, a traditional parser generator is used which also generates Java code. A component can reuse other components by simply including their abstract grammars, behavior, and context-free grammars into the generation process. Figure 3 illustrates this generation architecture. The generated Java code is placed in two local packages: `AST` and `parser`.

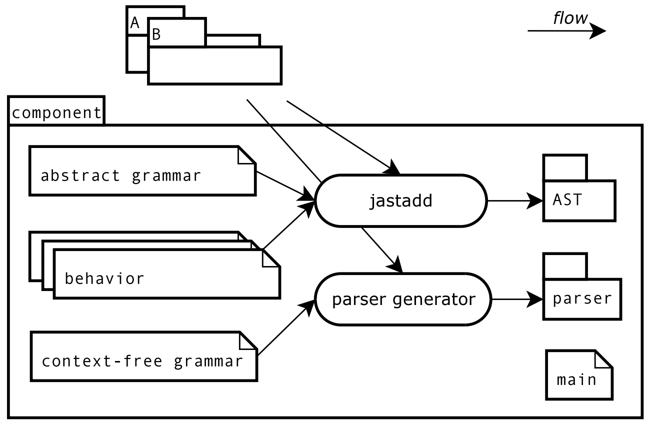

**Figure 3.** Generation architecture

## 2.4 Modularization

The abstract and context-free grammars can be modularized in a rather simple way, drawing on their similarity to object-oriented class hierarchies that can be extended with new subclasses. The real key to the extensibility of *JastAddJ* is the way the *behavior* can be specified in a declarative way, using the constructs in JastAdd.

The modular extension of Java 1.4 into Java 5 is a very challenging case. In particular, *generics* in Java 5 crosscuts aspects of compilation like name and type analysis in an intricate way. Due to the declarativeness of JastAdd, these aspects can be described as separate modules.

Because of the declarativeness, specification order is irrelevant, and the reason for grouping a set of attributes and equations together into a module is to promote reuse and understandability. JastAddJ is decomposed at two levels. At the coarse-grained *component* level, the decomposition is done based on what large components (directories) that are expected to be reused, e.g., a Java 1.4 frontend. At the finer-grained *module* level, a component is decomposed into individual modules (files), each consisting of a set of attributes, equations, etc. This decomposition is based primarily on understandability.

Figure 4 shows the decomposition of the Java 1.4 and Java 5 frontend components into modules, and illustrates how different criteria can be used for the modularization. The Java 1.4 frontend is decomposed according to the different kinds of analyses performed in the compiler, such as name analysis, type analysis, definite assignment, etc. This decomposition is largely based on the structure of the Java language specification (JLS) [GJSB00]. For example, name analysis is described in section 6 in the JLS. The Java 5 frontend is instead decomposed according to the new language constructs: the enhanced-for statement, autoboxing, etc. The module for enhanced-for describes how all the analyses (name analysis, type analysis, etc.) are extended in order to handle the enhanced-for construct. Because of the declarative specification, many other decompositions would have been possible as well.

| *Java 1.4 frontend* | *LOC* | *Java 5 extension* | *LOC* |
|---|---|---|---|
| Abstract grammar | 261 | | 47 |
| Behavior | | Behavior | |
| Name analysis | 2 481 | Enhanced for | 65 |
| Type analysis | 1 387 | Autoboxing | 197 |
| Definite assignment | 1 054 | Static imports | 110 |
| Exception handling | 208 | Generics | 2 394 |
| Constant expressions | 467 | Varargs | 141 |
| Anonymous classes | 124 | Enums | 339 |
| Class files | 475 | Annotations | 369 |
| Unreachable statements | 127 | | |
| Prettyprinter | 788 | | |
| Misc | 659 | | |
| Bytecode reader | 1157 | | 689 |
| Context-free grammar | 1053 | | 538 |
| Main program | 111 | | 20 |
| *total* | *10 352* | | *4 909* |

**Figure 4.** Modules in the Java 1.4 and 5 frontends

## 3. Design principles

The JastAdd system allows the behavior of compilers to be specified *declaratively*, using equations and other declarative constructs. While declarative, these constructs can easily be understood in terms of normal object-oriented programming, and the result of a JastAdd specification is an object-oriented framework that can be either used as is, by normal Java code, or it can be further extended using JastAdd constructs. In this section we will discuss these design ideas and how they have been applied in *JastAddJ*.

### 3.1 The attributed AST

A program is represented inside the compiler as an abstract syntax tree (AST). The tree nodes are objects of AST classes that are generated from the abstract grammar which defines both a class hierarchy and a composition hierarchy. For example, `While` is defined as a subclass of `Statement`, and as having two children: one `Expression` for the condition and one `Statement` for the body.

Compilation problems are cast into problems of defining *attributes* of the AST nodes. For example, the goal of name analysis is to associate each use of an identifier with the appropriate declaration, according to the scope rules of the language. This problem is cast into the problem of defining a `decl` attribute of identifier access nodes, such that the value is a reference to the appropriate declaration node. This problem can be further decomposed into subproblems that are also defined by attributes. For example, the scope rules that define the visibility of declarations can be defined using additional attributes.

Attributes that are references to AST nodes are called *reference attributes* [Hed00], and make many problems straightforward to express declaratively. This is in contrast to traditional Knuth-style attribute grammars [Knu68] that usually result in very cumbersome specifications for other than very simple problems.

While the attributes are defined declaratively by equations in JastAdd, they are accessible to ordinary programmers as methods on the AST classes. For example, the attribute `decl` will be represented as a method `Declaration Access.decl()` where `Access` and `Declaration` are AST classes modelling identifier accesses and declarations respectively.

## 3.2 Graphs superimposed on the AST

The AST tree structure gives a basic hierarchical representation of the program, traversable through methods generated as part of the AST class hierarchy, for example, `getCondition()` and `getBody()` for accessing the children of a `While` node. For many compilation problems, additional graph structures are useful, for example inheritance graphs, method call graphs, etc. The use of reference attributes, like `decl`, results in a superimposed graph structure on the AST.

Such graphs may well be cyclic. Consider two mutually dependent classes `A` and `B`, where `A` has a variable of type `B` and `B` has a variable of type `A`. The resulting graph, considering both child edges and the `decl` reference attributes, is cyclic. In practice, many graphs are cyclic, so it is very useful that the declarative underlying system allows them to be expressed.

Specific graphs can be defined by defining additional attributes. For instance, suppose we want to define a type graph, capturing the subtype relation. The essential information for this graph can be found in the `decl` attributes of the *extends* and *implements* clauses in the class declarations. An explicit graph can be defined by adding an attribute `supertypes` to the ClassDeclaration, defined as the set of declarations referred to in the extends and implements clauses.

## 3.3 The AST as the only data structure

The use of an attributed AST, and using reference attributes to superimpose graphs onto the AST, makes it straightforward to use the AST as the only data structure. For example, instead of using the separate symbol tables of traditional compilers, the corresponding information is cast as attributes in the AST. This is useful, because it allows the extension mechanisms that apply to ASTs and attributes to be applied to the compilation data structures too.

JastAdd attributes can have *parameters*, making it possible to define suitable APIs to the AST. For example, instead of using the traditional symbol tables for looking up declarations of identifiers, many AST nodes are equipped with an attribute `Declaration lookup(String identifier)` that will return the appropriate declaration that is visible at that particular node. Through tree edges and reference attributes, subproblems can be delegated to other nodes and their attributes. For example, the `lookup` attribute for an ordinary identifier access may be defined in terms of other lookup attributes in the enclosing class and its superclasses.

## 3.4 Declarative frameworks

The generated result of a JastAdd specification is an object-oriented framework: a set of collaborating AST classes with a method API that is generated from the attribute definitions. The framework can be used as is by client Java code: constructing an AST by instantiating the concrete AST classes (typically using a parser), and calling the attributes in the API to query the AST for information. For example, we could query the `decl` attribute of an identifier access in order to find its declaration.

The framework can also be extended with new behavior. Such extensions are made in JastAdd, resulting in a generated extended Java framework. The extensions can be made both along the syntactic dimension, adding new AST classes, and along the behavior dimension, adding new attributes to the AST classes.

The framework is *declarative* in that the behavior is defined through equations. From the usage perspective this means that once an AST has been constructed (for example, via a parser), all its attributes automatically have values such that all equations hold. The client code does not have to worry about in which order the attributes are given values; this is part of the implementation of the attribute methods and is automatically generated by JastAdd. In fact, this evaluation is done on-demand, as attribute values are used, and many values are cached for efficiency.

A consequence of the declarative definition of the attributes is that it is safe for client code to use any attributes and in any order. There are no hidden assumptions on the order in which the attributes may be called: their values are constant for any given AST. For this reason all attributes are represented as public Java methods in the resulting framework. Nevertheless, when writing a behavior module, there are typically some attributes that are intended to be used by clients, others that are intended to be defined when extending the language, and yet others that are simply help attributes that solve small subproblems. Informally, we therefore speak of a *client interface* and an *extension interface* to the different behavior modules. The client interface simply consists of classes with methods. The extension interface makes use of a number of declarative extension mechanisms.

## 3.5 Declarative extension mechanisms

JastAdd combines ordinary object-oriented programming extension mechanisms with some declarative mechanisms especially targeting tree based computations. They all share the common purpose to simplify the use of the major design ideas presented so far.

Language structure is specified by an *object-oriented abstract grammar* from which JastAdd generates a Java class hierarchy including constructors and traversal API. This provides the usual object-oriented opportunities for abstraction through deep class hierarchies with late bound methods and reuse through inheritance. *Inter-type declarations* are used to allow modular extension to an existing class hierarchy. JastAdd takes an extreme approach where no behavior is specified directly in the abstract grammar, but always through external behavior modules that introduce new attributes and equations in AST node classes.

The main mechanism to specify extensible behavior modules is the use of declarative attributes whose values are defined by equations. The equations are specified in a syntax directed fashion, solving a problem for each AST node type in isolation. The order of specifying the equations is irrelevant: an attribute evaluation engine automatically combines the equations into a global solution, ordering the evaluation of individual equations. This makes it trivial to combine modules—no manual code is needed for ordering the computations.

*Synthesized attributes* are very similar to virtual methods without side-effects. An attribute is specified in a class and equations may be overridden in subclasses. *Inherited attributes* propagate information about the current context downwards in a tree while decoupling the use of an attribute from its definition. The node reading an attribute value need not be aware of which node defines that value but only that there is an ancestral node providing an equation. *Circular attributes* can be used when there are cyclic dependencies between equations which are then evaluated using fixed-point iteration.

When using the AST as the only data structure it is crucial to be able to incrementally add new information to the AST to represent data that is not available at parse time. *Nonterminal attributes* are attributes that are subtrees defined using attributes and grafted into the existing AST. These attributes may make use of other attributes and provide a means to define new trees as functions of an existing tree. JastAdd also supports conditional *rewrites* that may use attributes to define context-dependent transformations to the AST. This can, for instance, be used to rewrite the AST into a form more suitable for later computations.

### 3.6 The JastAdd specification language

As a background to the examples in the upcoming sections, we briefly introduce the JastAdd specification language through a simple example language.

The language contains an abstract class A, and four concrete classes B, C, D, and E. Classes D and E are subclasses to A. Class D has two children called myB and myC, of types B and C respectively. B, C and E have no children.

```
abstract A;
B;
C;
D: A ::= myB:B myC:C;
E: A;
```

A declares a synthesized attribute sa of type int and with a default value 42. D overrides the default with an equation defining sa to equal 4711. The attribute and the equation are introduced into the classes A and D using inter-type declarations:

```
syn int A.sa() = 42;
eq D.sa() = 4711;
```

B declares an inherited attribute ia of type int. E declares an inherited attribute r of type C. Because C is an AST class, r is a reference attribute. D declares a nonterminal attribute myNta of type E, and provides a default value, new E():

```
inh int B.ia();
inh C E.r();
nta E D.myNta() = new E();
```

The inherited attributes must be defined in an ancestor node. D defines the ia of its myB child using the sa attribute of its myNta nonterminal attribute. D also defines the r of myNta as equal to its myC child reference:

```
eq D.myB().ia() = myNta().sa() + 1;
eq D.myNta().r() = myC();
```

Figure 5 shows a class diagram for the language. The inter-type declarations have been moved into the appropriate classes. The aggregate relations show the AST hierarchy.

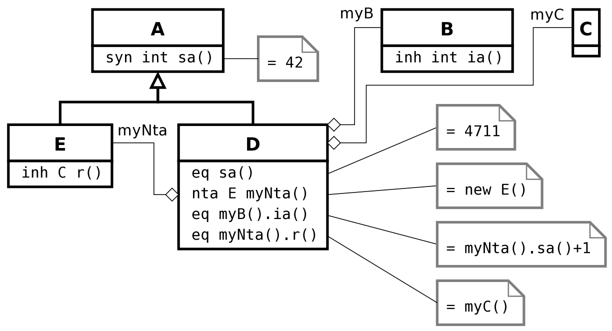

**Figure 5.** Class diagram for example language

Figure 6 shows an attributed AST for the language. The unattributed AST with the D, B, and C nodes is created by, for example, a parser. The attribution, including the E object and the r, ia, and sa attributes, is then computed automatically by the attribute evaluator.

## 4. Name Analysis

The goal of name analysis is to bind each access of an identifier to its corresponding declaration. While the implementation is intricate, the resulting client interface in *JastAddJ*

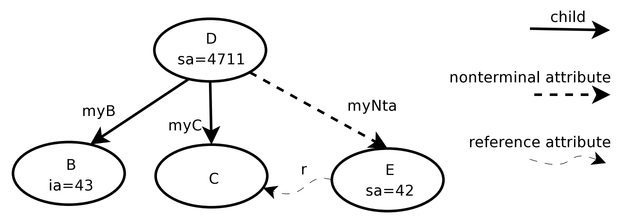

**Figure 6.** Attributed AST for example language

is very simple, as shown in Figure 7. It consists of the single attribute `decl` in `Access` nodes.

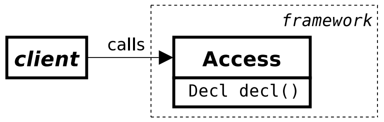

**Figure 7.** Client interface to the name analysis

Java name analysis contains many challenging and interesting problems. While there are many subtleties in the individual language constructs, there are two main subproblems. First, the visibility rules need to handle different combinations of scopes, such as nested scopes, inheritance, and qualified access to remote members. Second, determining the meaning of a name is fairly complex: it depends on the syntactic context and it may also depend on the resolution of other names in that context.

In a previous article we have shown how name analysis is solved in our Java 1.4 frontend [EH06]. In this section we will show how this solution forms two extension interfaces, one for each of the main subproblems, and we will exemplify how these interfaces can be used when extending the language.

## 4.1 Visibility rules

The set of visible identifiers at an AST node is defined by an attribute `lookup(`*`string`*`)` which returns the appropriate visible declaration node. `Access` nodes use the `lookup` attribute to define their `decl` attribute. The `lookup` attribute is declared as inherited, meaning that its value is defined by an ancestor node.[1] By declaring an attribute as inherited, we say that the node *captures* the value defined by the ancestor.

Figure 8 shows the extension interface to the visibility rules. In the framework there are some classes that take on the role of *defining lookup* for their children. This is done by an equation that defines the `lookup` attribute for a given child of the node. The equation holds for all `lookup` attributes in the complete subtree of the child, but may be redefined further down in the subtree, in another node that defines lookup. Such redefinition can be used, for example, to model nested scoping. A language construct that affects visibility typically redefines `lookup` for its children. It does so by capturing its own `lookup` attribute and combining it with other *specific lookups*. The specific lookups are synthesized attributes that model some specific visibility. Examples include looking up names in a given block, in a given class, in the inheritance chain of a given class, and so on.

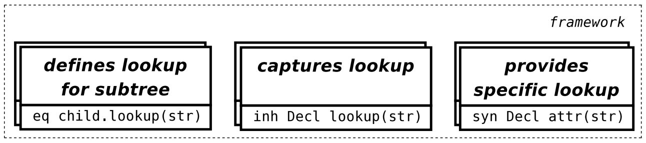

**Figure 8.** Extension interface to the visibility rules

Classes in the framework take on the roles of defining lookup, capturing lookup, and providing specific lookups in order to implement the visibility rules of the different Java 1.4 language constructs. The visiblity definition for fields, parameters, and local variables, includes 14 classes taking on the role of define lookup, 7 classes that capture lookup, and 9 classes that provide specific lookup. The corresponding numbers for types are 9 classes that define lookup, 5 classes that capture lookup, and 5 classes that provide specific lookup. The reason that type visibilty requires fewer rules than variable visibilty is that there are fewer language constructs that may enclose type declarations than variable declarations. Consequently, method visibilty requires even fewer rules. The same three roles can be used also when extending the framework, as will be exemplified in Section 4.3.

## 4.2 Determine the meaning of names

Names in Java are highly context-sensitive and a Java parser typically builds general `Access` nodes for all names, regardless of their actual meaning. The JLS defines the specific rules for how to first classify context-free names according to their syntactic context and then to refine them by reclassifying contextually ambiguous names. Our Java compiler follows this implementation scheme [EH06]. While the details of this implementation are quite intricate, the extension interface is very simple: a new language construct that has an `Access` child needs only provide an equation defining its initial `NameKind` which can be a type, a package, an expression, or ambiguous (either a package or a type, depending on other context). The further refinement of ambiguous names is then carried out automatically by the Java 1.4 frontend.

Figure 9 shows the extension interface to determining the meaning of names. The framework declares the inherited attribute `nameType` for `Access` nodes. A new language construct that has an `Access` child needs to provide an equation for this attribute.

## 4.3 Extensions

The complete name analysis framework consists of the two parts described in sections 4.1 and 4.2. Using this simple

[1] In a traditional attribute grammar [Knu68], the value of an inherited attribute is defined by the immediate parent node. JastAdd uses a shorthand, similar to the Eli *including* feature [KW94], so that it is sufficient if there is a definition of the attribute somewhere along the ancestor spine of the AST.

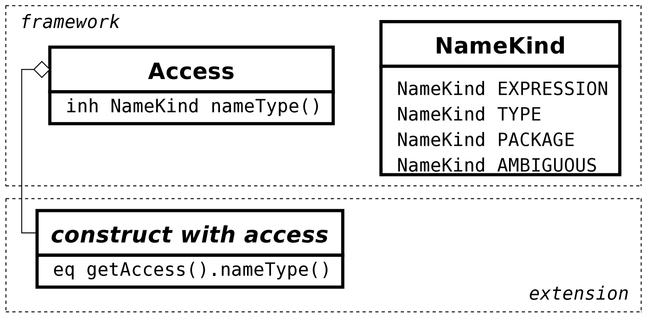

**Figure 9.** Extension interface to determining the meaning of names

framework it is very easy to define new language constructs that extend Java with constructs that affect the name analysis.

### 4.3.1 The enhanced for loop

Consider extending Java 1.4 with the *enhanced for* loop of Java 5:

`for(Element e : collection)` *`statement`*

A local variable declaration `e` is declared and needs to be included in the set of visible declarations of the contained statement. The new loop is modelled by a new AST class `EnhancedFor`, see Figure 10. It defines lookup for its contained statement by an equation that delegates to a new specific lookup, `matchLocal`. The specific lookup first matches the string with the local variable, and if no match, it delegates to the `EnhancedFor`'s own `lookup` attribute, which is captured by the superclass `Stmt`.

The abstract syntax for the `EnhancedFor` is

`EnhancedFor: Stmt ::= Access Decl Exp Stmt`

Since `EnhancedFor` is a construct with an `Access` child (`Element` in the example above), it also provides an equation to define the `nameType` of that `Access` as equal to `NameKind.TYPE`.

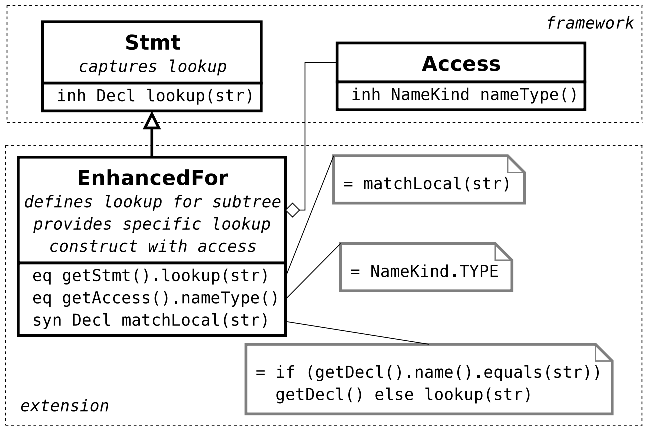

**Figure 10.** Defining name analysis for the enhanced for loop

### 4.3.2 Adding an inspect statement

As another example, consider extending Java with an *inspect* statement in which the members of the inspected object are visible without qualification. This new statement changes the set of visible declarations for its contained statement:

`inspect(obj)` *`statement`*

The new statement is modelled by a new AST class `InspectStmt`. Similarly to the previous example on the `EnhancedFor`, the `InspectStmt` defines the `lookup` attribute of its contained statement, see Figure 11.

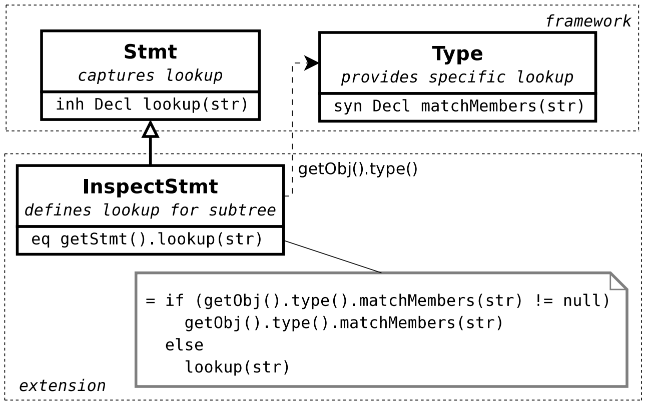

**Figure 11.** Defining visibility in the inspect statement

The equation reuses the existing specific lookup attribute `matchMembers` of the inspected object's type. A simple combination of this attribute with the current visibility context, i.e., the `lookup` attribute of the `InspectStmt` itself, yields the desired visibility context for the contained statement. This simple equation will cause all `Access` nodes inside the inspected statement to be automatically bound to the appropriate declarations.

The example illustrates the reuse of specific lookup attributes in the name analysis framework. The whole Java 1.4 name analysis framework is in fact built up incrementally using this technique: some language constructs introduce new specific lookup attributes, others make use of these attributes to define visibility. For example, Method declarations and Class declarations provide specific lookup attributes. These are combined in various ways in different language constructs to support block nesting, inheritance, nested classes, and qualified access to type members.

The example also illustrates that name analysis and type analysis are mutually dependent: the definition of the `lookup` attribute of the `InspectStmt` uses the type analysis framework as well (see the next section): it uses the `type` attribute of its inspected object. Type analysis on the other hand requires name binding to find the type of a name and to bind type names to declarations. Name binding and type analysis are thus mutually dependent and would require complex manual scheduling if not using declarative attributes.

## 5. Type analysis

The two main tasks of the type analysis framework are to compute the type for expressions and to decide whether two types are in the subtype relation. The way these analyses are implemented is highly affected by the design principle to use the AST as the only data structure which is the key to their extensibility. This requires types to be represented by nodes in the AST, and the type of an expression is represented simply by a reference attribute that points to the appropriate type node. We will use the term *binding* as a synonym to a reference attribute, in particular when the attribute binds together distant parts of the AST.

This design leads to the simple client interface shown in Figure 12. The attribute `type` binds each `Expr` to a `TypeDecl` which can then be used to compute the subtype relation using the `subtype` attribute.

Extending the language with new kinds of types then boils down to the following two problems. First, to obtain AST representations for the new types. Second, to extend the subtype test so that the instances of the new types can be compared with each other and with other existing types.

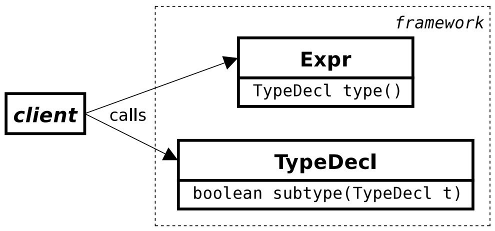

**Figure 12.** Client interface to the type analysis

### 5.1 Type representation

Java has both explicitly and implicitly declared types. For explicitly declared types, like classes and interfaces, we use their declaration nodes as type representations. For implicitly declared types, for example primitive types and arrays, we add AST nodes as part of the attribute evaluation, i.e., after parsing. This is done declaratively through the use of nonterminal attributes. A nonterminal attribute is a child node that is defined by an equation, rather than constructed by the parser. In JastAdd, nonterminal attributes are evaluated on demand, i.e., the nonterminal attribute nodes are constructed automatically as soon as they are accessed.

Figure 13 shows the extension interface for type representation. In the framework there are abstract classes like expressions and declarations that *bind to a type*. If the extension introduces subclasses to these, called *concrete binds to type* in the figure, they have to supply an equation for binding to the desired type node. The type node can be an instance of an existing `TypeDecl` subclass in the framework, or of a *new type* in the extension. Typically, the equations defining type bindings make use of the name analysis framework, i.e. the `decl` attribute, to find the desired type object.

Suppose the extension introduces new types. In case the extension also introduces explicit declarations of these types, the nodes can be automatically built by the parser. But if the new types are implicitly declared, they need to be built as nonterminal attributes. In this case they are built as children to another AST class, a *declaration context*. The use of intertype declarations allows an extension to add attributes and equations to existing classes in the framework.

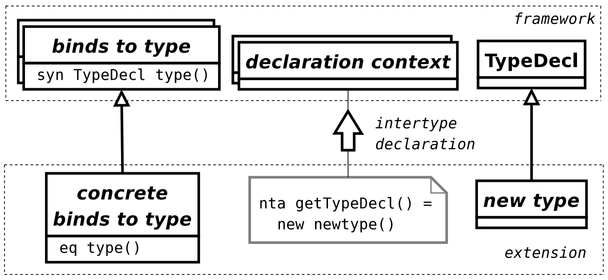

**Figure 13.** The extension interface for type representation

### 5.2 Extensible subtype tests

The subtype relation is the foundation of object-oriented type checking and used when implementing other relations such as assign conversion, method invocation conversion, casting conversion, etc. We implement the subtype relation through a parameterized attribute `boolean TypeDecl.subtype(TypeDecl t)` to determine if two types are in the subtype relation.

All kinds of types are possible to compare using the subtype attribute. Not only class and interface types, but also primitive types and array types. To compare class and interface types, we use a straightforward implementation of the subtype test that searches the direct supertypes transitively. Since all types are represented by nodes in the AST, we can simply follow the reference attributes that bind type declarations to their direct supertypes.

The `subtype` attribute supports comparison between two arbitrary types. To allow the modular definition and extension of this relation, we have used the double dispatch pattern [Ing86].[2] When comparing two types by `t1.subtype(t2)`, the `subtype` attribute dispatches on `t1`, and its definition uses another attribute that dispatches on `t2`. The combination of double dispatch with inter-type declarations allows extensions to be done modularly, as illustrated below.

The double dispatch pattern allows us to specify the subtype relation with a single equation for each pair of type kinds. In practice, the number of equations is much smaller since many types are incompatible, and these combinations can rely on default equations that simply return *false*.

[2] Binary attributes, analogous to binary methods [BCC$^+$95], could have been used as an alternative mechanism for achieving extensible implementation, but are not supported by JastAdd.

Figure 14 shows the extension interface for subtyping. The `subtype` attribute is a synthesized attribute in the abstract class `TypeDecl`. Each different kind of type is represented by a subclass to `TypeDecl` and provides an equation for `subtype`. The framework includes a number of such types, here exemplified with `T1` and `T2`. For each such type, say `T1`, there is also an attribute `supertypeOfT1` in `TypeDecl`, with the default value `false`.

The framework can be extended by adding a new type, say `NewType`, as a subclass of `TypeDecl`. Its equation for `subtype` should do the double dispatch, i.e., it should call `supertypeOfNewType`. The declaration of this attribute is added to the abstract `TypeDecl` in the framework by means of an inter-type declaration, also with `false` as a default value. The `NewType` then overrides this equation and provides the implementation for comparing two `NewTypes`. If a `NewType` can be a supertype of another type, say `T1`, in the framework, then an equation for `supertypeOfT1` is added to `NewType` to capture this relation. And conversely, if a `NewType` can be a subtype of a `T1`, then an equation for `supertypeOfNewType` should be added to `T1`, again using an inter-type declaration. As seen in Figure 14, the use of inter-type declarations allows the extension to be expressed modularly: even the additions to the existing classes in the framework can be expressed inside the extension module.

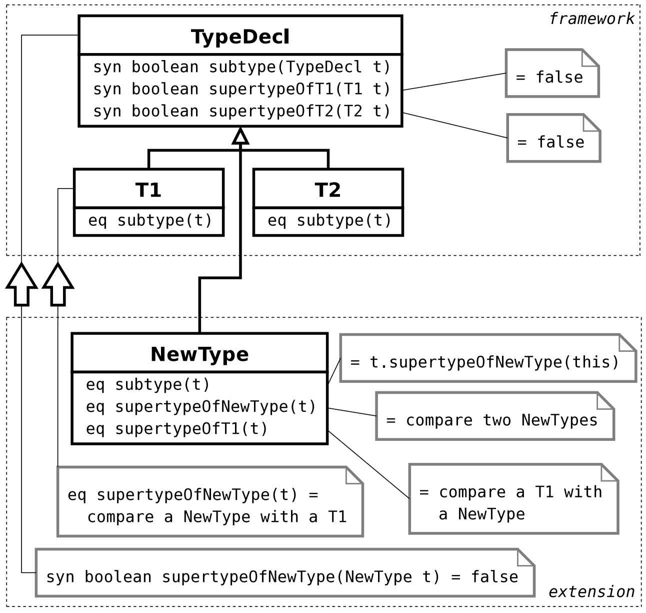
framework
TypeDecl
syn boolean subtype(TypeDecl t)
syn boolean supertypeOfT1(T1 t)
syn boolean supertypeOfT2(T2 t)
= false
= false
T1
eq subtype(t)
T2
eq subtype(t)
NewType
eq subtype(t)
eq supertypeOfNewType(t)
eq supertypeOfT1(t)
= t.supertypeOfNewType(this)
= compare two NewTypes
= compare a T1 with a NewType
eq supertypeOfNewType(t) = compare a NewType with a T1
syn boolean supertypeOfNewType(NewType t) = false
extension

**Figure 14.** The extension interface for subtyping

## 5.3 Extensions

### 5.3.1 Non-null types

As an example extension, consider the addition of non-null types to Java, to prevent null pointer exceptions. We have implemented such an extension, including non-null type inference for legacy code [EH07], and supporting virtual dispatch during object initialization using the concept of raw types [FL03].

The type system is extended to distinguish between *possibly-null* types, and guaranteed *non-null* types. The Java 1.4 framework includes the AST class `ClassDecl` which models possibly-null types. For each `ClassDecl` node in the AST, there should be a non-null counterpart. This is modelled by a new class `NonNull`. Instances of `NonNull` are added as nonterminal attributes of `ClassDecl`. An attribute `possiblyNull` is added to `NonNull` that points back to its parent `ClassDecl`. This makes it easy to go back and forth between a possibly-null type and its non-null counterpart, as needed in other equations. See Figure 15.

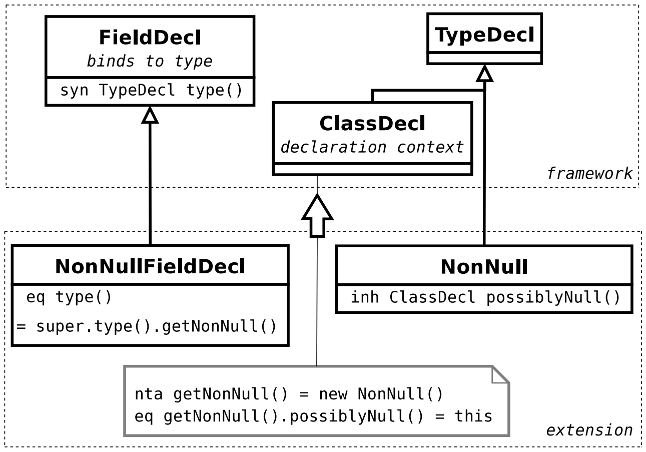

**Figure 15.** Extending the type representation framework with non-null types

Equations that extend the subtype relation are shown in Figure 16. The framework includes, in addition to `ClassDecl`, a class `NullType` that models the type of the null value. The extension includes two boilerplate equations for the double dispatch. Of the two remaining equations, one handles the comparison between two non-null types. This is computed by comparing the types between their possibly-null counterparts, using the `possiblyNull` attribute defined earlier. The last equation is added by means of an inter-type declaration to `ClassDecl`, and compares the `ClassDecl` with a `NonNull` type, by delegating the comparison to the possibly-null counterpart of the latter. The converse comparison, checking if a `NonNull` type is a supertype of a `ClassDecl`, will always be false, and is covered by the default equation. No additional equations are needed for comparing `NonNull` with `NullType` because these are never in the subtype relation and are covered by the default equations.

To handle raw types, the type analysis framework is extended in a similar way.

### 5.3.2 Generic types

Java 5 generic types are implemented by extending the Java 1.4 framework in a similar way as for non-null types. Each

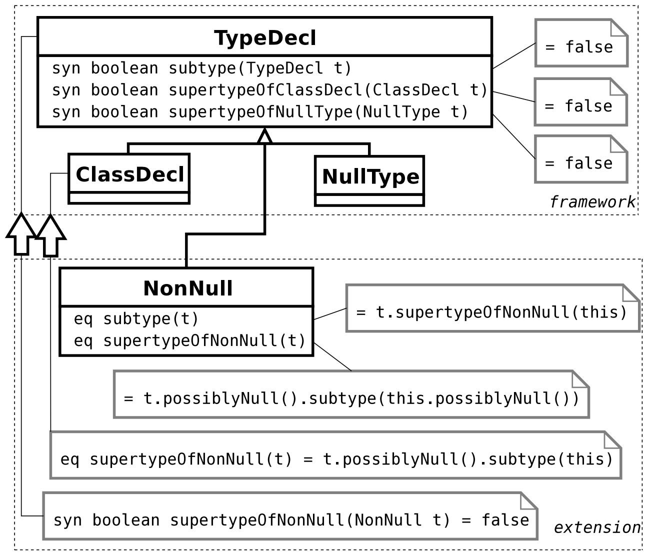
TypeDecl
syn boolean subtype(TypeDecl t)
syn boolean supertypeOfClassDecl(ClassDecl t)
syn boolean supertypeOfNullType(NullType t)
= false
= false
= false
ClassDecl
NullType
framework
NonNull
eq subtype(t)
eq supertypeOfNonNull(t)
= t.supertypeOfNonNull(this)
= t.possiblyNull().subtype(this.possiblyNull())
eq supertypeOfNonNull(t) = t.possiblyNull().subtype(this)
syn boolean supertypeOfNonNull(NonNull t) = false
extension

**Figure 16.** Extending the subtyping framework with non-null types

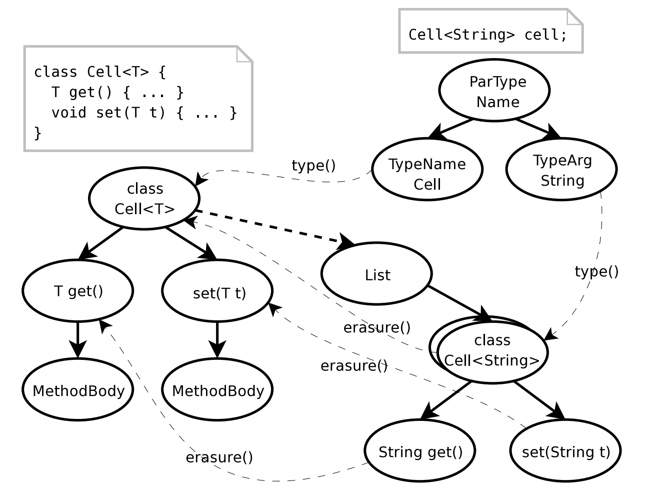

**Figure 17.** Simplified AST for representation of generic types

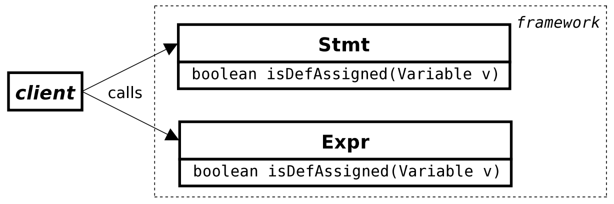
framework
client
calls
Stmt
boolean isDefAssigned(Variable v)
Expr
boolean isDefAssigned(Variable v)

**Figure 18.** Client interface to definite assignment

parameterization of a generic type is built using a nonterminal attribute. Figure 17 shows a simplified AST for a generic class `Cell` and a parameterization using `String` as the type argument. The nonterminal attribute is depicted as a dashed tree edge.

A main difference, as compared to the non-null example, is that each generic class may have many parameterizations. The nonterminal attribute therefore takes a list of arguments. Specific field and method signatures are also built as part of the nonterminal attribute, where the type parameter `T` has been replaced by the type argument `String`. This way, much of the other compilation, such as name lookup and qualified access to members of parameterized types, need not be aware of generic types.

We also provide access to the erased version of a type and its members through the `erasure()` reference attributes.

## 6. Definite assignment

The JLS prescribes that each local variable and every blank final field must have a *definitely assigned* value prior to any access to its value. A Java compiler must therefore carry out a conservative flow analysis to make sure that each control flow path to a variable access contains at least one assignment to that variable. A similar computation, *definitely unassigned*, is needed to ensure that a final variable is only assigned at most once.

The client interface in Figure 18 provides information whether a variable is definitely assigned or not in a particular context. Since variable assignment is an expression in Java, the flow analysis needs to propagate information not only through statements but also through expressions.

Figure 19 shows the extension interface for definite assignment which is centered around the `isDAbefore` and `isDAafter` attributes, both parameterized with the desired `Variable`. Each language element that *contains* children that are part of the execution flow needs to provide an equation for the inherited attribute `isDAbefore`. This attribute represents the status before the element is executed. Each *execution element* needs to provide an equation for the status after execution by defining the `isDAafter` attribute.

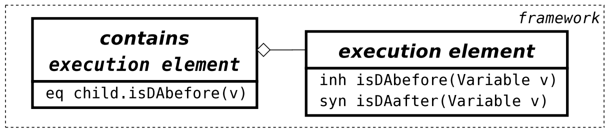

**Figure 19.** Extension interface to definite assignment

These mechanisms for handling definite assignment are used inside the framework, and can also be used in extensions of the framework. Figure 20 shows how the `While` statement is handled, assuming for a moment that it is defined in an extension rather than inside the framework. The abstract grammar for `While` is the following.

```
While: Stmt ::= cond:Expr body:Stmt;
```

Notice that the `While` is both an execution element itself (as are all statements), and it contains other execution elements: `cond` and `body`. The informal specification of Definite Assignment in Chapter 16 of the JLS states that:

- A variable is definitely assigned before the condition if it is definitely assigned before the while statement.
- A variable is definitely assigned before the loop body if it is definitely assigned after the condition.
- A variable is definitely assigned after a WhileStatement if it is definitely assigned after the condition.

Notice the similarity between this syntax-directed way of specifying definite assignment and the equations in Figure 20.

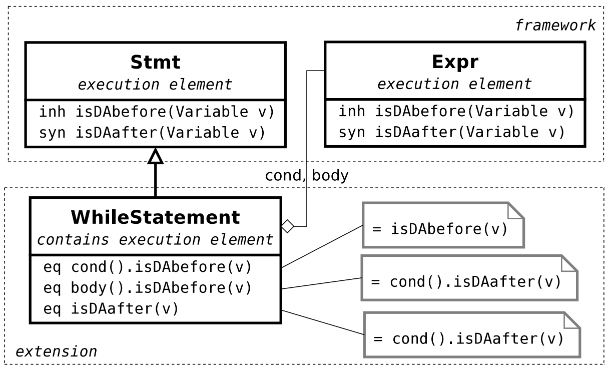

**Figure 20.** Extension of the definite assignment framework for `While`

Definite unassignment provides additional challenges in that there may be circular dependencies between attributes as shown by the following example. A variable is only definitely unassigned before a loop condition in a while statement if it is definitely unassigned both before the while statement and after the loop body. Moreover, whether a variable is definitely unassigned after the loop body depends on whether the variable is definitely unassigned before the loop condition. This leads back to the initial definition which leads to a circularity. Such attributes can be expressed declaratively using circular attributes in JastAdd. The evaluation is carried out using a fix-point computation, starting at a start value specified in the declaration of the circular attribute. Because the computation is declaratively expressed, it is straightforward to extend it for new language constructs, simply by supplying additional equations for the new language construct, analogously to how it was done for the `While` statement in Figure 20.

## 7. Discussion

The previous sections have demonstrated three key declarative frameworks that can be extended modularly when adding new language constructs. There are additional frameworks with similar small extension interfaces in *JastAddJ* that deal with *unreachable statements*, *control flow*, *constant expressions*, *error checking*, and *code generation*. Each new language construct needs to extend some of these frameworks in order to support error checking and/or class file generation. For instance, the *inspect statement*, used earlier to exemplify name analysis extension, would also need to extend several of the other frameworks to fully extend the compiler. New analyses that are added to the compiler can benefit from using these frameworks and may also define their own extended reusable frameworks.

Section 3 presented the major design principles that enable modular extensions to *JastAddJ*. The syntax directed approach to declaratively specify context-sensitive computations using attributes provides a foundation for extensible computations. Attributes allow specifications to be broken down into small problems that are solved for each kind of node type separately, and then combined into a whole automatically by the attribute evaluation engine. Synthesized attributes enable abstraction over the node children while inherited attributes enable abstraction over the current context. Traditional use of symbol tables break these qualities by introducing unnecessary dependencies between unrelated computations that make use of the tables. Mutually dependent analyses must then be scheduled manually as discussed in the previous sections, e.g., object-oriented name and type analysis. The extension mechanisms all rely on tree based computations and the design principle to use the AST as the only data structure serves to maintain that property throughout the compiler.

There are three main language features in JastAdd that enable the use of the AST as the only data structure while maintaining a loose coupling between different parts of the AST. First, references to remote nodes allow for abstraction over distant structures. Attributes may be accessed remotely through a reference without exposing unnecessary details about the remote context. Second, inherited attributes with parameters enables abstraction over the current context without introducing further dependencies besides that there must be an ancestor that defines the value in the current context. This kind of abstraction occurs naturally in programming languages where one can often refer to enclosing elements implicitly, e.g., break out of the current loop, or explicitly, a named variable currently in scope. Third, nonterminal attributes can be used to build new subtrees as functions of existing structures during attribute evaluation which is important to represent implicitly declared entities that need to be explicit in the AST.

## 8. Evaluation

We have evaluated *JastAddJ* by comparing it to a number of open source Java compilers, both extensible and non-extensible ones. The evaluation has been done from several perspectives: language compliance, performance, and implementation size. We first present the compilers we compare with, then the test suites we have run, and finally the results of the different comparisons.

All tests were carried out on a ThinkPad T42p Pentium M 1.7 Ghz with 1 Gbyte memory, running Fedora Core 6 Linux. Unless otherwise noted the tests were executed using the Java HotSpot Client VM version 1.5.0_02-b09 and we report the fastest of five runs.

### 8.1 Compilers overview

We include the most common industrial strength Java compilers as well as a few research compilers that explicitly support extensibility. If a compiler exists in both a Java 1.4 and a Java 5 version we include both, the reason being that we want to use the latest version for compliance comparisons but the Java 1.4 version when comparing implementation size. All compilers are implemented in Java unless otherwise stated. We have only included compilers that are publicly available including source code. We chose not to include the GNU compiler for Java (gcj) since they are currently replacing their frontend with the eclipse frontend, which is already included in the comparison.

**Javac** Javac is the standard compiler in Sun JDK and also serves as a reference implementation for the Java Programming Language. It is implemented in Java and in the comparison we include javac 1.4.2 and javac 1.5.0.

**Eclipse** The Eclipse project contains an incremental Java compiler based on technology evolved from the VisualAge Java compiler. We have tried to remove all IDE specific compiler code to allow for a fair comparison to the stand alone compilers. Version 2.1.3 supporting Java 1.4 only and version 3.1.2 that also supports Java 5 are included in the comparison.

**Jikes** Jikes is a high-performance Java 1.4 compiler written in C++ originally developed by IBM at T. J. Watson research Center but now maintained by an open source community. The comparison is based on version 1.22.

**Polyglot** Polyglot is framework for building Java frontends using a library that can be extended with new language features and analyses through inheritance [NCM03]. Version 2.2.0 of polyglot is included in the comparison. We have also included an experimental Java 5 extension, version R20061211, which is a modular extension to Polyglot 1.3.4 [SM07].

**JaCo** JaCo is an extensible compiler for Java 1.4 written in a slightly extended Java dialect called Keris that supports extensible algebraic data types with defaults [jac07b, ZO01a]. Keris is itself implemented as a modular language extension to JaCo.

### 8.2 Test suites

The most widely used test suites for Java are probably the SPECjvm98 and SPECjbb2000. They are not particular suitable for evaluating our extensible compiler since they focus on testing JVMs rather than if the compiler can handle all language idiosyncrasies. The Java Compatibility Kit (JCK) is an extensive test suite that can be licensed from Sun to ensure compatible implementations of Java. This is an excellent test suite perfectly targeting our domain. However, the open license is read-only which prevents us from automated processing of the test suite. Visual inspection reveals that it is an excellent test suite with many challenges both for static semantic analysis and code generation including Java 5 features. We therefore have to rely on alternative test suites to validate the correctness of the compilers. We have used the Jacks test suite, the DaCapo benchmark, and a number of sample applications, as described below.

#### 8.2.1 Jacks

Jacks is an excellent compiler killing test suite that validates the static semantic analysis performed by Java 1.4 compilers using a test suite that mimics the chapters from the JLS, Second Edition. It was originally developed at IBM but is now maintained by the Mauve project. The suite does unfortunately neither test Java 5 features nor run-time behavior of the generated code.

#### 8.2.2 DaCapo benchmark

The DaCapo benchmark is a set of general purpose, realistic, freely available Java applications combined with an evaluation methodology for benchmark suites and a performance evaluation methodology [BGH$^+$06]. While the main goal of the DaCapo suite is not to test the frontend part of a compiler it provides an automated test harness and result validator that can be used to verify that the code generated by the compiler is correct. We have used DaCapo release dacapo-2006-10-MR2 which includes the following applications in its test suite:

**antlr** A parser generator and translator generator.

**bloat** A byte-code level optimization and analysis tool.

**chart** A graph plotting toolkit and pdf renderer.

**eclipse** An integrated development environment.

**fop** An output-independent print formatter.

**hsqldb** An SQL relational database engine.

**jython** A python interpreter.

**luindex** A text indexing tool.

**lusearch** A text search tool.

**pmd** A source code analyzer.

**xalan** An XSL processor.

#### 8.2.3 Sample applications

The previous benchmarks only test Java 1.4 features. There are, to our knowledge, no open test suites for Java 5 which enable the same systematic testing. We therefore handpicked a few applications that make extensive use of Java 5 features. We also included older Java 1.4 versions of the same applications which allow us to compare how the language extensions affect compilation. The packages we included from the JDK are particularly challenging in that they make extensive use of tricky language features such as nested classes, anonymous classes, and generic types using wildcards. Finally we included two substantial applications to verify that the compilers scale to moderate sized applications in the range of 100K LOCs.

**junit 3.8.1** JUnit testing framework, 3.6K LOC of Java 1.4 source.

**junit 4.1** JUnit testing framework, 3.1K LOC of Java 5 source.

**jhotdraw 5.3** JHotDraw GUI framework for structured graphics, 14.6K LOC of Java 1.4 source.

**jhotdraw 7.0.8** JHotDraw GUI framework for structured graphics, 38.7K LOC of Java 5 source.

**JDK 1.4.2** The java.lang and java.util packages from JDK 1.4.2, 35.7K LOC of Java 1.4 source.

**JDK 1.5** The java.lang and java.util packages from JDK 1.5, 56.2K LOC of Java 5 source.

**ecj 3.2.2** The Eclipse Java compiler, 94.1K LOC of Java 1.4 source.

**Jigsaw** The official W3C Java-based Web server, 100.8K LOC of Java 1.4 source.

## 8.3 Compiler compliance

We compiled the Jacks test suite on all selected Java compilers. Warnings were turned off not to produce warnings about common programming mistakes which are not considered errors in the test suite, e.g., a warning that a finally block can not complete normally. The compilers that support Java 5 were run in Java 1.4 compatibility mode. The results are shown in Figure 21 showing number of passed, skipped, and failed tests. The tests that were skipped are tests that cause the compilers to get stuck in a never ending loop. The skipped tests for javac, eclipse, and *JastAddJ1.4* are all caused by the same bug in the standard class library where a floating point number is converted from a string representation into its binary counterpart. Although *JastAddJ1.4* passes more tests than any of the other compilers, we do not claim superiority to either compiler but merely conclude that the number of failed test cases indicates that the compiler implements the complete static semantic analysis for Java 1.4.

While the Jacks test suite does an excellent job in testing language idiosyncrasies we use another set of benchmarks to validate our thesis that most language features are used by even fairly small real world applications. Figure 22 shows the success rate for a set of sample programs, described in Section 8.2.3, ranging from a few thousand to more than 100K LOCs excluding comments and whitespace. Both jaco and polyglot give numerous false positives and internal compiler errors which shows that real applications indeed contain many of the language details tested by the Jacks test suite.

| Compiler | % pass | # pass | # skip | # fail |
|---|---|---|---|---|
| javac1.4 | *99.0 %* | 4446 | 1 | 44 |
| javac1.5 | *99.2 %* | 4455 | 1 | 35 |
| eclipse1.4 | *98.1 %* | 4409 | 1 | 81 |
| eclipse1.5 | *98.6 %* | 4429 | 1 | 61 |
| jikes | *99.3 %* | 4461 | 0 | 30 |
| polyglot2 | *90.5 %* | 4065 | 49 | 377 |
| jaco | *78.0 %* | 3505 | 3 | 983 |
| *JastAddJ1.4* | *99.5 %* | 4468 | 1 | 22 |

**Figure 21.** Results from running the Jacks test suite. Tests that are skipped cause the compiler not to terminate.

| Compiler | junit | jhotdraw | JDK | ejc | jigsaw |
|---|---|---|---|---|---|
| javac1.4 | √ | √ | √ | √ | √ |
| javac1.5 | √ | √ | √ | √ | √ |
| eclipse1.4 | √ | √ | √ | √ | √ |
| eclipse1.5 | √ | √ | √ | √ | √ |
| jikes | √ | √ | √ | √ | √ |
| polyglot2 | √ | √ | fail | fail | fail |
| jaco | √ | fail | fail | fail | fail |
| *JastAddJ1.4* | √ | √ | √ | √ | √ |

**Figure 22.** Results from compiling the Java 1.4 applications described in Section 8.2.3.

The results from compiling the Java 5 applications described in Section 8.2.3 are shown in Figure 23. We have only included the Java 5 enabled compilers since it is rather pointless compiling Java 5 code with a Java 1.4 compiler. It is interesting to notice how challenging it is to compile even quite small Java 5 applications. One of the major reasons is the extensive use of generics and wildcards in the collection framework. Even quite simple usage of these classes may for instance rely on inference of type parameters for generic methods.

| Compiler | junit 4.1 | jhotdraw 7.0.8 | JDK 1.5 |
|---|---|---|---|
| javac1.5 | √ | √ | √ |
| eclipse1.5 | √ | √ | √ |
| polyglot5 | fail | fail | fail |
| *JastAddJ5* | √ | √ | √ |

**Figure 23.** Results from compiling the Java 5 applications described in Section 8.2.3.

The DaCapo test suite is used to check that the compilers generate correct code and not only perform static semantic analysis. Figure 24 shows the execution time of the benchmarks in the suite for code generated by javac, eclipse, and *JastAddJ1.4* . The compilers perform virtually no optimizations but rely on dynamic optimization in the virtual machine. The exact layout of bytecode may still have significant effects since the HotSpot compiler is highly optimized for the particular layout performed by javac. We notice that there is still some room for improvement in the *JastAddJ1.4* backend for *bloat* and *xalan* but that the performance is very similar for the rest of the benchmarks.

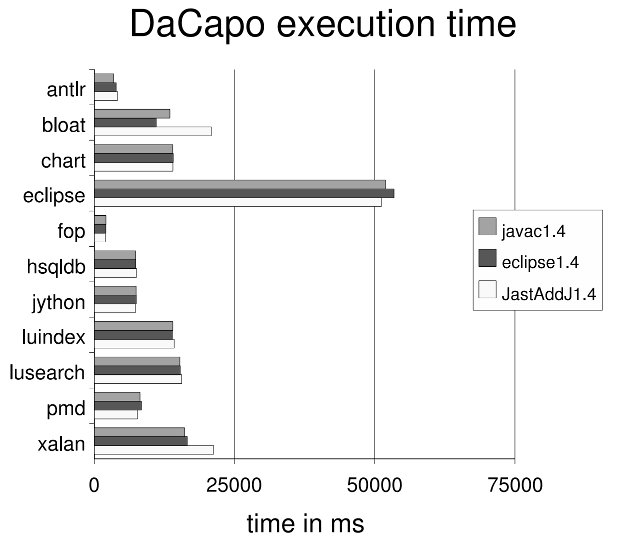

**Figure 24.** Execution time for the DaCapo benchmark suite.

## 8.4 Compilation time

To evaluate the speed of our generated compiler we have compared compilation times for the applications described in Section 8.2.3. Each application was compiled five times and the shortest compilation time for each Java 1.4 application is shown in Figure 25. Jikes is by far the fastest compiler which is not that surprising since it is implemented in C++ and does not suffer from VM startup time and initial compilation. The results indicate that our generated compiler is less than three times slower than the fastest java based compiler. We also notice that *JastAddJ* is several times faster than polyglot, another extensible compiler, on medium sized applications.

The compile times for the Java 5 applications is shown in Figure 26. The generated *JastAddJ5* based compiler is less than three times slower than javac which is the fastest Java 5 compiler. *JastAddJ5* is actually narrowing the gap to javac1.5 compared to the coresonding Java 1.4 implementations. We believe this stems from the demand-driven creation of parameterized types used in *JastAddJ5* where parameterized body declarations are only built when being used by client code. This shows that using the AST as the only data structure is feasable for even quite complex language extensions. It is also worth noticing that we do not perform any analysis or optimization of the grammars. We believe that there are plenty of opportunities for domain-specific optimizations in this area related to caching and evaluation strategies. That would hopefully further narrow the gap between JastAdd generated compilers and hand written compilers.

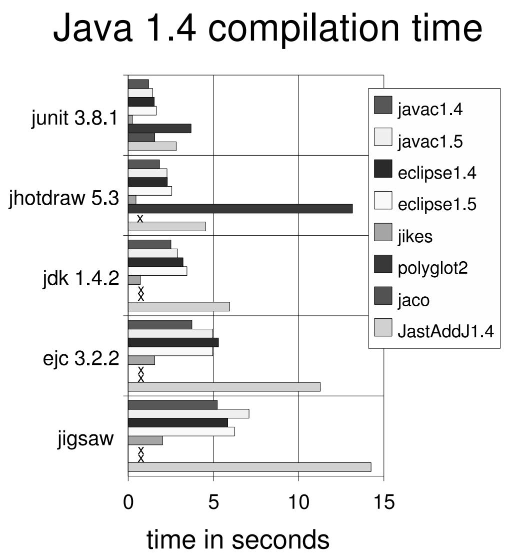

**Figure 25.** Compile time for Java 1.4 applications.

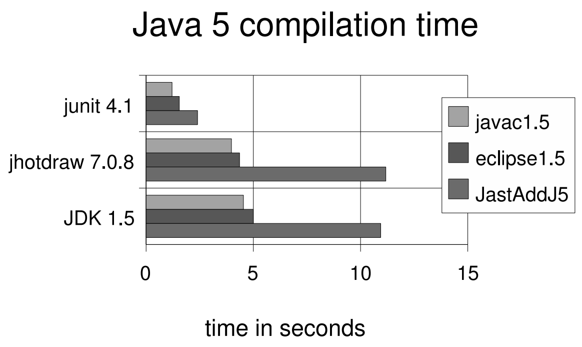

**Figure 26.** Compile time for Java 5 applications.

## 8.5 Implementation size

The tested compilers are implemented in different language dialects or sometimes even completely different languages. We still find it interesting to compare implementation sizes to get a rough estimate of the implementation effort, and we have used SLOC-count to count the number of lines of code, excluding comments and whitespace [Whe07]. We have also included the number of tokens since the number of lines can differ quite substantially depending on coding style. Figure 27 shows the various sizes for the tested compilers. We are somewhat surprised by the large differences in source size, often being more than twice as large as the javac compiler. The Eclipse compiler is an incremental compiler normally used within an IDE which may account for the larger code size. *JastAddJ1.4* and JaCo stand out as being significantly smaller than Javac. Both compilers use extended versions of Java especially targeting compiler implementation which seems to pay off. However, the poor compliance results for JaCo makes it hard to draw any definite conclusions. The JastAdd based implementation shows that the declarative specification technique not only allows for extensibility but also yields significantly smaller compilers. The lower half of the comparison contains the Java 5 compilers. *JastAddJ5* and Polyglot have modular extensions that can be enabled at will while javac and eclipse are later generations of the compiler. The *JastAddJ5* Java 5 extension is roughly 5K LOC of code while the Polyglot extension is 24K LOC, and even compared to the base compiler size, the extension for *JastAddJ5* is proportionally smaller than the corresponding Polyglot extension. Moreover, doing the same comparison for the modular extension and the delta between javac1.4 and javac1.5 shows that they are proportionally about the same size. This indicates that not only is JastAdd based compilers smaller than handwritten compilers in Java but the modular extensions scale in the same way as in-place changes to the code base.

| Compiler | # KLOC | # KTokens |
|---|---|---|
| javac1.4 | 21 ( *100 %* ) | 106 ( *100 %* ) |
| eclipse1.4 | 57 ( *271 %* ) | 288 ( *271 %* ) |
| jikes | 70 ( *333 %* ) | 342 ( *322 %* ) |
| polyglot2 | 39 ( *186 %* ) | 220 ( *207 %* ) |
| jaco | 16 ( *76 %* ) | 73 ( *68 %* ) |
| *JastAddJ1.4* | 15 ( *71 %* ) | 58 ( *54 %* ) |
| javac1.5 | 30 ( *100 %* ) | 155 ( *100 %* ) |
| eclipse1.5 | 83 ( *297 %* ) | 411 ( *265 %* ) |
| polyglot5 | 63 ( *210 %* ) | 340 ( *219 %* ) |
| *JastAddJ5* | 21 ( *66 %* ) | 87 ( *56 %* ) |

**Figure 27.** The source code size of the compilers using javac as baseline. The upper half compares Java 1.4 compilers while the lower part compares Java 5 compilers.

## 9. Related work

Traditional compilers may provide well-engineered APIs for adding additional analyses, e.g., the JDT model in the Eclipse Java compiler [ecl07], but they are often less suited for language extensions. For instance, the ajc compiler for AspectJ [asp07] is an excellent integration of AspectJ extensions and the Eclipse Java compiler, but the integration is non-modular and requires manual synchronization of the two code bases.

There are Java source-to-source translators that provide support for extensions at the syntactic level but that do not support extensible static-semantic analysis, e.g., JavaBorg/MetaBorg [BV04], the Java Syntactic Extender [BP01] and the Jakarta Tool Suite [SB02]. These tools translate an extended Java dialect to pure Java and rely on a separate compiler for the actual compilation to bytecode. While this approach is attractive for its simple implementation it has serious drawbacks when it comes to handling context-sensitive information. The translation can not include context-sensitive properties such as the type of an expression in the translation strategies. Since the approach is based on source-to-source, a separate Java compiler is needed to perform error checking and bytecode generation. Error checking is performed on the generated code, and errors are rarely well aligned with the original source code.

There are also approaches that provide support for static-semantic analysis but more limited support for syntactic extensions. OpenJava [TCIK00] adds a macro system to Java that uses a meta-object protocol (MOP), similar to Java's reflection API, to manipulate the program structure. Macro programs can access data structures representing a logical structure of a program from which much of the semantic structure of the program is exposed. The MOP can be used to add additional analyses on top of Java but there is little support for refining existing analyses or for syntactic extension.

The most flexible solution for language extensibility is to provide support for extensions at both the syntactic and static-semantic analysis level. Polyglot is an extensible source-to-source compiler framework implemented in Java that relies on design patterns for extensibility, e.g., abstract factories, extensible visitors based on delegation, and proxies [NCM03]. The base code is a Java 1.4 frontend which has been extended successfully for numerous language features. The frontend has for instance been extended with the AspectJ language in the AspectBench project [ACH$^+$06]. The extension is modular and uses the Soot optimization framework as a backend to form a full AspectJ compiler [VRHS$^+$99]. The compiler is pass oriented (with extensible passes) and also supports tree rewriting at the end of each pass.

JaCo is an extensible Java 1.4 compiler including both frontend and backend [ZO01b]. The first implementation of JaCo was done in a Java dialect supporting algebraic types with defaults [ZO01a]. A set of object-oriented architectural patterns was used to further support extensibility. JaCo has later been implemented in Keris, an extension to Java that supports extensible modules with explicit refinement and specialization mechanisms. Both compilers are based on explicit scheduling of multiple passes.

The above compilers all rely on manual scheduling of dependencies between analyses. In contrast, the JastAdd Extensible Java compiler is implemented in the declarative ReRAGs formalism, combining the language mechanisms automatically while supporting modularity and extensibility as described in the previous sections. ableJ is a declarative

Java frontend implemented using the Silver attribute grammar system [WKSB07]. The system supports full Java 1.4 concrete syntax and limited semantic analyses such as type checking. While certainly useful for experimenting with language features the lack of support for complete Java 1.4 is a major drawback when trying to evaluate language features on real programs rather than small examples. Only a few experimental Java 5 extensions are available and there are no published performance measurements or systematic testing.

## 10. Conclusions

We have presented *JastAddJ*, the JastAdd Extensible Java compiler, and demonstrated that it is a practical high-quality extensible compiler: large programs can be handled, the specifications are smaller than hand-crafted code, and highly non-trivial modular extensions can be developed, like adding generics to a language. The compilation is slower than in hand-crafted compilers, within a factor of three of Javac. This is still very reasonable when considering the main application area: to build special-purpose analyses and to build extended languages. Our compiler outperforms all other extensible compilers for Java that we know of, considering all measures: Java compliance, compilation speed, specification size, and the support for non-trivial extensions. We have also shown that our compiler can handle large programs, over 100K LOCs, and that our modular extensions scale in the same way as in-place changes to the code base of a non-extensible compiler.

As a side-result of our experimental evaluation, we have come to the interesting conclusion that a Java compiler needs to be extremely compliant with the language specification in order to be of much practical use: compilers that do not handle almost all of the special cases of the language constructs will fail on all except a few small programs. This is bacuse even using the JDK libraries requires advanced features such as inference of type parameters for generic methods.

To demonstrate that the technique is indeed applicable to practical problems of extending Java, we have built substantial extensions to our Java 1.4 compiler. This includes an extension to Java 5, an AspectJ frontend, an alternative backend that compiles to Jimple, and a pluggable non-null type checker.

In implementing *JastAddJ*, we have developed a number of general design techniques for building extensible compilers, and techniques for dealing with central aspects of compilation, such as name analysis, type analysis, and definite assignment. We have illustrated how these highly non-trivial compilation problems can be expressed as declarative frameworks with small extension interfaces, that allow straightforward modular extension. In doing so, we have used a number of declarative extension mechanisms from the attribute grammar field, such as inherited attributes, reference attributes, nonterminal attributes, and circular attributes. We think that this illustrates that these features blend very well with object-oriented programming.

The design techniques we have developed are general and can be reused for building extensible compilers for other languages as well. For example, a current effort involves building an extensible compiler for a physical modelling and simulation language, Modelica [ÅEH07]. In principle, the modules that define the core declarative frameworks could be refactored into separate components and reused for many languages. However, these modules are so small that such implementation reuse is hardly worth the effort. We find it more important to document the ideas as we have done in this paper, so that the ideas themselves can be reused.

## Acknowledgments

We thank David Notkin and the anonymous reviewers of OOPSLA for helpful comments on earlier drafts of this paper.

## References

[ACH+06] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. *Transactions on Aspect-Oriented Software Development*, 1(1), 2006.

[And05] Per Andersson. *Efficient modelling and synthesis of data intensive reconfigurable systems*. PhD thesis, Lund University, Sweden, June 2005.

[asp07] ajc in the AspectJ project, 1.5.0, 2007. http://www.eclipse.org/aspectj/.

[BCC+95] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[BGH+06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of OOPSLA'06*. ACM Press, October 2006.

[BP01] Jonthan Bachrach and Keith Playford. The Java syntactic extender (JSE). In *Proceedings of OOPSLA'01*, pages 31–42. ACM Press, 2001.

[Bra04] Gilad Bracha. Pluggable Type Systems. In *OOPSLA'04 workshop on revival of dynamic languages*, 2004.

[BV04] Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions.

In *Proceedings of OOPSLA'04*, pages 365–383. ACM Press, October 2004.

[ecl07] Eclipse Java Compiler, Eclipse Java Development Tools 3.1.2, 2007. http://download.eclipse.org/eclipse/downloads/drops/R-3.1.2-200601181600/index.php.

[EH04] Torbjörn Ekman and Görel Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004*, volume 3086 of *LNCS*, pages 144–169. Springer, 2004.

[EH06] 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.

[EH07] 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.

[Ekm06] Torbjörn Ekman. *Extensible Compiler Construction*. PhD thesis, Lund University, Sweden, June 2006.

[FL03] M. Fahndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of OOPSLA'03*, pages 302–312, 2003.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[Hed00] Görel Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), pages 301–317, 2000.

[HM03] Görel Hedin and Eva Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[Ing86] D. H. H. Ingalls. A simple technique for handling multiple polymorphism. In *Proceedings of OOPSLA'86*, pages 347–349, 1986.

[jac07a] The Jacks compiler test suite, 2007. http://sources.redhat.com/mauve/.

[jac07b] JaCo Java Compiler, The Programming Language Keris, 2007. http://lampwww.epfl.ch/ zenger/keris/.

[jas07] JastAdd, 2007. http://jastadd.cs.lth.se/web/.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP 2001*, volume 2072 of *LNCS*, pages 327–355. Springer, 2001.

[Knu68] 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).

[KW94] Uwe Kastens and William M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31(7):601–627, 1994.

[MH03] E. Magnusson and G. Hedin. Circular Reference Attributed Grammars - Their Evaluation and Applications. *Electr. Notes Theor. Comput. Sci.*, 82(3), 2003.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Compiler Construction, 12th International Conference, CC 2003*, volume 2622 of *LNCS*, pages 138–152. Springer, 2003.

[Nil06] Anders Nilsson. *Tailoring native compilation of Java for real-time systems*. PhD thesis, Lund University, Sweden, May 2006.

[Pal07] Palpable Computing - a new perspective on Ambient Computing, 2007. http://www.ist-palcom.org.

[SB02] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[SM07] Milan Stanojevic and Todd Millstein. Java 5 extension for Polyglot compiler framework, 2007. http://www.cs.ucla.edu/ milanst/projects/polyglot5/.

[TCIK00] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. OpenJava: A Class-Based Macro System for Java. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering*, pages 117–133. Springer, 2000.

[VRHS$^+$99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[VSK89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of PLDI '89*, pages 131–145. ACM Press, 1989.

[Whe07] David A. Wheeler. SLOCCount, 2007. http://www.dwheeler.com/sloccount/.

[WKSB07] Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. Attribute Grammar-based Language Extensions for Java. In *Proceedings of ECOOP'07*, LNCS. Springer, 2007.

[ZO01a] Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of ICFP'01*, pages 241–252. ACM Press, 2001.

[ZO01b] Matthias Zenger and Martin Odersky. Implementing extensible compilers. In *Workshop on Multiparadigm Programming with Object-Oriented Languages*, Budapest, Hungary, June 2001.

[ÅEH07] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Development of a Modelica compiler using JastAdd. In *Proceedings of LDTA'07. Electr. Notes Theor. Comput. Sci.*, 2007.