Extensible Compiler Construction

Torbjörn Ekman

Doctoral dissertation, 2006

Department of Computer Science
Lund University

ISBN 91-628-6839-X
ISSN 1404-1219
Dissertation 25, 2006
LU-CS-DISS:2006-2

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: `tobbe@ekman.net`

Typeset using LaTeX 2ε

Printed in Sweden by Tryckeriet i E-huset, Lund, 2006

© 2006 by Torbjörn Ekman

Abstract

Processing of programs is a core area in computer science. A compiler that translates source text to machine language is the most well-known kind of tool in this area, but there are numerous other kinds of related applications: source-to-source translators, refactoring tools, reengineering tools, metrics tools, consistency checkers, etc. These tools perform similar analyses and can therefore benefit from shared infrastructure. This thesis addresses the problem of how to build program-processing tools much more easily, providing high-level concise ways of programming the tools, and providing good modularity and extensibility, allowing for a high degree of reuse between tools.

We present Rewritable Reference Attributed Grammars (ReRAGs), a technique that builds on well-known software development techniques such as object-orientation, aspect-oriented software development, declarative programming, attribute grammars, and transformation systems. ReRAGs combine mechanisms from these areas into one coherent framework with synergistic effects on modularity and extensibility. These mechanisms support several different decomposition criteria for modularization that enable re-use: as *separate computations* on the program model, as a base language and *language extensions*, and the same decomposition as used in a *language specification* for traceability. ReRAGs allow such modules to be decoupled from each other and automatically resolves complex context-sensitive dependences.

An evaluation algorithm for the formalism is presented and implemented in the JastAdd tool which combines ReRAGs with Java. We have implemented a complete Java 1.4 compiler to demonstrate the full potential of the JastAdd system and to evaluate its mechanisms for modularization and extensibility. We show how name analysis for Java with its complex visibility rules involving nested scopes, inheritance, qualified access, and syntactic ambiguities can be modularized in the same way as the informal Java language specification. We have also extended our Java compiler with non-null types for detection of possible null-pointer violations at compile time. The extension is completely modular and the technique allows for so called pluggable type systems that can be enabled at will.

The techniques scale to real languages and large applications. Our generated Java compiler passes as many tests as production use compilers during compliance testing, compiles applications larger than 100.000 lines of code, and the executable specification is less than two-thirds the size of handwritten compilers.

Acknowledgements

The work presented in this thesis has been carried out within the *Software Development Environments* group at the Department of Computer Science, Lund University. I would like to thank my supervisors, Dr. Görel Hedin, Dr. Klas Nilsson, and Professor Boris Magnusson, for giving me the freedom to pursue research directions that I have found interesting. I am particularly grateful to my main supervisor Görel Hedin. Much of the work presented in this thesis is joint work with her.

I would also like to thank Ulf Hagberg at ABB Automation Technology Products Malmö and Klas Nilsson for many rewarding discussions on the Control Modules language and automation technology in general.

The are a few people that I would like to thank for nice ideas, joint work, and discussions on various topics: Anders Nilsson for numerous compiler related discussions and for using my Java front-end in his Java2c compiler, Eva Magnusson for her work on CRAGs and initial implementation of JastAdd, Roger Henriksson and Sven Gestegård Robertz for various GC related discussions, Ulf Asklund and Lars Bendix for joint work on agile configuration management, and David Svensson for rewarding discussions within the PalCom project.

Christian Andersson and Anders Ive deserve credit for showing me the very different world outside the university, and I'm equally indebted to the Wednesday breakfast club for putting things into perspective.

Finally, I am very grateful to Karin Wanhainen for her love, support, and late night attribute grammar discussions. You will always be the root in my abstract syntax tree.

This work has been performed within the Center for Applied Software Research (LUCAS) at Lund University, and is partially funded by ABB Automation Technology Products, the PalCom integrated project in EU's 6th Framework Programme, and VINNOVA, the Swedish Agency for Innovation Systems.

List of Papers

The research papers included in this thesis are:

I. Torbjörn Ekman and Görel Hedin. Rewritable Reference Attributed Grammars. *Proceedings of ECOOP 2004: 18th European Conference on Object-Oriented Programming*, Oslo, Norway, June 2004.

II. Torbjörn Ekman and Görel Hedin. The JastAdd System - modular extensible compiler construction. *Submitted for publication*, 2005.

III. Torbjörn Ekman and Görel Hedin. Benchmarking the JastAdd Extensible Java Compiler. *Unpublished manuscript*, 2006.

IV. Torbjörn Ekman and Görel Hedin. Modular name analysis for Java using JastAdd. *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering*, Braga, Portugal, LNCS, Springer, 2006. To appear.

V. Torbjörn Ekman and Görel Hedin. Pluggable non-null types for Java. *Unpublished manuscript*, 2006.

Papers related to the JastAdd project not included in this thesis

- A. Nilsson, A. Ive, T. Ekman and G. Hedin, Implementing Java Compilers Using ReRAGs. *Nordic Journal of Computing*, Vol 11:3(213-234), 2004.
- T. Ekman, Design and Implementation of Object-Oriented Extensions to the Control Module Language. *In proceedings of the 11th Nordic Workshop on Programming and Software Development Tools and Techniques - NWPER 2004*, Turku, Finland, 2004.
- T. Ekman and G. Hedin. Reusable Language Specification Modules in JastAdd II. *Position paper at ERLS 2004, Workshop on Evolution and Reuse of Language Specifications for Domain-Specific Languages*, Oslo, 2004.
- T. Ekman and G. Hedin. Automatic renovation of Java programs using ReRAGs - examples and ideas. *Position paper at 5th International Workshop on Object-Oriented Reengineering (OOR 2004)*, Oslo, 2004.
- T. Ekman, A case study of Separation of Concerns in Compiler Construction using JastAdd II. *Proceedings of third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Lancaster, UK, March, 2004.

Other peer-reviewed papers

- T. Ekman and U. Asklund, Refactoring-aware versioning in Eclipse. *Electr. Notes Theor. Comput. Sci.*, Vol 107, 2004.
- U. Asklund, L. Bendix, and T. Ekman, Software Configuration Management Practices for eXtreme Programming Teams. *In proceedings of the 11th Nordic Workshop on Programming and Software Development Tools and Techniques - NWPER 2004*, Turku, Finland, 2004.
- U. Asklund, L. Bendix, and T. Ekman, Configuration Management for eXtreme Programming. *In proceedings of the Third Conference on Software Engineering Research and Practise in Sweden*, Lund, Sweden, October 23-24, 2003.
- A. Nilsson, T. Ekman, and K. Nilsson, Real Java for Real Time – Gain and Pain. *In Proceedings of CASES-2002*, pages 304-311. ACM, ACM Press, October 2002.
- A. Nilsson and T. Ekman, Deterministic Java in Tiny Embedded Systems. *In Proceedings of The Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC-2001)*, pages 60-68. IEEE Computer Society, May 2001.

Contents

Extensible Compiler Construction 1
1 Introduction 1
2 Challenges in extensible compilers 2
3 Background 5
4 Contributions 16
5 Conclusions and future work 19

I Rewritable Reference Attributed Grammars

Rewritable Reference Attributed Grammars 29
Torbjörn Ekman, Görel Hedin
1 Introduction 29
2 Typical examples of AST rewriting 31
3 Background 33
4 Rewrite rules 35
5 ReRAG evaluation 41
6 Implementation algorithm 45
7 Implementation evaluation 51
8 Related work 53
9 Conclusions and Future Work 56

II The JastAdd System

The JastAdd System – modular extensible compiler construction 61
Torbjörn Ekman, Görel Hedin
1 Introduction 61
2 Application domain 62
3 Features and Foundation 63
4 The JastAdd evaluation engine 68
5 Related tools and systems 68
6 Conclusions 71

III Benchmarking the JastAdd Extensible Java Compiler

Benchmarking the JastAdd Extensible Java Compiler 77
Torbjörn Ekman, Görel Hedin
1 Introduction 77
2 Java compilers background 78
3 Compiler compliance 80
4 Compilation time 80
5 Implementation size 82
6 Conclusions 83

IV Modular name analysis for Java using JastAdd

Modular name analysis for Java using JastAdd 87
Torbjörn Ekman, Görel Hedin
1 Introduction 87
2 JastAdd Background 88
3 Name analysis for DemoJavaNames 91
4 Related work 101
5 Conclusions 102

V Pluggable non-null types for Java

Pluggable non-null types for Java 107
Torbjörn Ekman, Görel Hedin
1 Introduction 107
2 Non-null types background 109
3 JastAdd Background 111
4 JavaDemoTypes Base language 114
5 Non-null types extension 122
6 Discussion 128
7 Automatic non-null type refinement 131
8 Evaluation 135
9 Related work 138
10 Conclusions and Future work 139

Extensible Compiler Construction

1 Introduction

Processing of programs is a core area in computer science. A compiler that translates source text to machine language is the most well-known kind of tool in this area, but there are numerous other kinds of related applications: source-to-source translators, refactoring tools, reengineering tools, metrics tools, consistency checkers, etc. These tools are usually complex and building them from scratch requires a major effort. However, the different tools perform similar analyses and can benefit from shared infrastructure. For example, a refactoring tool for Java needs to do much of the same analysis as is done by a Java compiler. And a compiler for a language similar to Java should be able to reuse much of an existing Java compiler implementation. Today, creating new languages or new tools for an existing language is so costly that such an effort is undertaken by few companies, and only when the new language or tool is to be heavily used. By implementing compilers and other program-processing tools in an extensible reusable manner, this could change. It might become affordable to build very special-purpose tools, intended to be used for a very limited purpose. For example, to do a special-purpose refactoring of a large body of legacy code. Another interesting prospect is easily to build domain-specific languages (DSLs) on top of existing general-purpose languages. Often, domain logic is captured in frameworks written in a general-purpose language. Turning such framework APIs into domain-specific language constructs can make programming more concise and easier to check statically for consistency.

This thesis addresses the problem of how to build program-processing tools much more easily, providing high-level concise ways of programming the tools, and providing good modularity and extensibility, allowing for a high degree of reuse between tools. We present Rewritable Reference Attributed Grammars (ReRAGs), a declarative technique for rewriting and analyzing abstract syntax trees in order to simplify compilation. An evaluation algorithm for the formalism is presented and implemented in the JastAdd tool which combines ReRAGs with Java. To demonstrate the full potential of the JastAdd system we implemented a full Java 1.4 compiler to evaluate its mechanisms for modularization and extensibility. We show how name analysis for Java with its complex visibility rules involving nested scopes, inheritance, qualified access,

and syntactic ambiguities can be modularized in the same way as the informal Java language specification [GJSB00,GJSB05]. We have also extended our Java compiler with support for non-null types for detection of possible null-pointer violations at compile time. The extension is completely modular and allows for pluggable type systems [Bra04], enabled at will. The generated compiler has been benchmarked against some popular Java compilers and passes at least as many tests in the Jacks test suite as production use compilers such as javac [jav06], the Eclipse Java compiler [ecl06], and jikes [jik06]. The compilation time is well within a factor of four compared to the handwritten compilers implemented in Java and the specification is only two-thirds the size of the smallest handwritten compiler.

The rest of this thesis introduction is structured as follows. Section 2 presents the challenges in extensible compiler construction addressed by this thesis. Section 3 presents the foundation on which ReRAGs is built and compares to related work. The contributions of this thesis are highlighted in Section 4 and Section 5 concludes the introduction and discusses future work.

2 Challenges in extensible compilers

While processing of programs is an old research topic, there are still plenty of opportunities for improvement when is comes to extensibility. This is particularly true for industrial tools that need to support full languages and process large bodies of code. Early stages in compilers such as scanning and parsing, are well understood and there exist generative techniques for extensible specifications [Vis97]. This thesis focuses on computations in later stages, e.g., name binding, type checking, and code optimization. Such computations rely on context-sensitive analysis and are still often hand coded in an ad-hoc fashion with little support for extensibility. The rest of this section describes the challenges addressed by this thesis.

2.1 Context-sensitive computations

Program-processing tools provide great challenges for extensible computations because of complex context-sensitive dependences in programming languages. In particular, the meaning of a name depends on its context, e.g., there should be a visible declaration that the name refers to. Object-oriented languages pose extra challenges in that the meaning of a name often depends not only on the immediate lexical context but also on other context-sensitive computations, e.g., the meanings of other names. A name may, for instance, refer to an inherited member, declared in a different class. The inherited class is, however, itself specified through a name. The single task of determining the meaning of

a name thus introduces complex dependences to other names in other classes. Such dependences need to be taken into account when implementing language extensions. If new constructs interact with names they may even introduce additional dependences between existing constructs.

To handle these complex problems, it is desirable to modularize computations and provide a uniform interface that abstracts from the implementation details. Other computations, e.g., metrics and refactoring tools, can then use such modules without knowing their internal details. This kind of abstraction is important from an understandability point of view since it allows complex problems to be decomposed into simpler ones. This also helps the developer to focus on one task at a time.

2.2 Modularization

When talking about code artifacts, the term *modularity* is often used to describe the possibility to decompose a system into modules [Par72]. Complex problems are usually decomposed in subproblems that are solved one at a time and then combined into a solution. The main purpose of modularity is to allow the code that solves each subproblem to be located in a separate module. This allows each subproblem, or concern, to be studied in isolation, a property which is known as *separation of concerns* [Dij82]. This is important from both an understandability point of view, i.e., to decompose the system into understandable modules, and from an extensibility point of view, i.e., to decompose the system into a base system and extensions.

The description of modules has been very abstract so far, using terms such as subproblem, code artifact, module, and concern to describe the individual parts of a system. The reason is that several different criteria can be used when decomposing a system into modules. One way to decompose a compiler is to divide it into separate *computations* that each perform one subproblem of the compilation process, e.g., scanning, parsing, static-semantic analysis, optimization, and code generation. The system is then divided into a basic model that represents the language constructs, and a set of computation modules that operate on that basic model. New computations, e.g., metrics, can then easily be added in a modular fashion while reusing existing semantic analysis. However, this kind of decomposition can be challenging when there are dependences between the various computations, in particular the context-sensitive computations described above.

Another way to decompose a compiler is into a base language and a set of *language extensions* that each provide additional functionality. This decomposition is useful when a language evolves and is extended with new language constructs, e.g., domain-specific constructs are added to a general-purpose language. The base language can then be composed with suitable extensions for

a particular domain, enabling a high degree of reuse between families of languages. However, it is likely that new language constructs affect both the language model and individual computations such as name binding, type analyses, and code generation. For instance, the enhanced *for statement* in Java 5 extends Java 1.4 with a new syntactic element and also refines the name binding module since it declares a new variable. There is thus a need to *extend* existing context-sensitive computations in a modular fashion besides the need to add syntactic elements and computations. It is worth noticing that the language constructs cross-cut the computation modules.

Programming languages are often defined in a formal or informal language specification. These specifications are rarely executable and a compiler needs thus to be implemented in a separate language. A third useful decomposition criterion is to use the same decomposition as in the *language specification* to provide traceability between the language specification and the executable implementation. Traceability is of great importance when debugging compilers. Programming languages are inherently complex and most compilers contain subtle errors that are hard to detect. These errors are easier to trace to the specification than to the implementation and the modularization helps locating the corresponding source location. The same arguments are equally valid for other kinds of specifications, e.g., formal proofs of type soundness or behavior preservation of refactorings.

To summarize we have the following decomposition criteria, all of which are desirable depending on the situation:

Separate Computations To decompose a compiler or similar tool into separate computations on the program model. The goal is to be able to reuse and combine the different computations to obtain different tools, and to express and understand each computation in isolation.

Language Extension To decompose a compiler or similar tool into an implementation of a base language and extensions to the language. The goal is to be able to reuse the base language implementation for many different extensions, to combine extensions, and to be able to express and understand each extension in isolation.

Language Specification To use the same decomposition of the compiler as is used for the informal language specification. The goal is to provide traceability between the language specification and the implemented compiler.

These criteria put challenging requirements on the used implementation language to allow the developer to modularize the compiler according to the preferred criteria. In particular, when considering the complex dependences in context-sensitive computations that occur naturally in programming languages. There is thus a need for implementation languages with strong support for separation of concerns.

2.3 Scalability

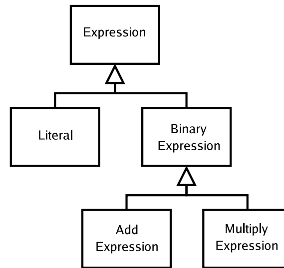
It is important that a compiler technology is scalable both to full languages and large sized applications to be of practical use. Mainstream programming languages are often complex and contain many corner cases. Research languages, on the other hand, tend to be small and pure at the cost of practical applicability. Our experience, from implementing a full Java compiler, tells us that a subset of Java is fairly easy to support but that the full language is much more challenging. This complexity manifests itself in that many research Java compilers pass significantly fewer tests in test suites than production use compilers. While a compiler for a subset of Java may be useful as a proof of concept for the compiler technology, it is less suited as a platform for code analysis and evaluation of language extensions. Class libraries and even fairly small applications tend to use most language features in Java and fail to compile if only a subset is supported. It is also worth noticing that for a tool to be useful in an industrial setting it needs to be able to analyze code bases in the range of a hundred thousand lines of code or more.

3 Background

The foundation of our approach to extensible compiler construction is the Rewritable Reference Attributed Grammars (ReRAGs) formalism developed as part of this thesis. It builds on several well-known software development techniques: object-orientation, aspect-oriented software development, declarative programming, attribute grammars, and transformation systems. ReRAGs combine mechanisms from these areas into one coherent framework with synergistic effects on modularity and extensibility. This section gives a background to these techniques and highlights important differences between our approach and other work on extensible compiler construction.

3.1 Object orientation

Object orientation provides a natural way of structuring data and computations. The data can be categorized in a class hierarchy where each class provides the state and behavior for a specific data element. A sample class hierarchy for expressions is shown below.



The use of a hierarchy enables multiple levels of abstraction. Polymorphism allows a reference of a certain class to not only reference instances of that class, but also instances of its subclasses. A suitable abstraction level, e.g., *Expression*, can then be used to hide internal details of expressions while a more concrete level, e.g., *BinaryExpression* or even *AddExpression*, can be used where more detail is needed. Common behavior can be placed in an abstract superclass, e.g., *Expression*, and then inherited to the concrete subclasses. Type checking can for instance often be shared by all *BinaryExpressions*. Inheritance with overriding allows for iterative refinement of behavior until the right level of detail is reached. Object orientation provides a nice modularization mechanism through the class concept that models both state and behavior in a single module that can also be extended in a modular fashion.

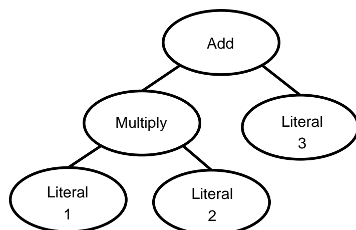
3.2 Abstract syntax trees

In the processing of programs, the typical core representation is a tree that captures the composition of language constructs in the program. Typically, an Abstract Syntax Tree (AST) is used, i.e., a tree where textual details have been abstracted away. Such trees can result from parsing a source file or from being synthesized directly. This thesis mainly deals with challenges in extensible analysis of programs represented by ASTs and relies on existing work in extensible parsing technology for building these trees [Vis97].

All possible forms of the ASTs can be described by an *abstract grammar* [McC64] that captures the basic rules for how different language constructs may be combined in a program. In many practical applications, e.g., compilers, the AST is often represented in an ad-hoc manner, and the computations are ordinary programs working on the ad-hoc representation. In our approach the ad-hoc AST representation is replaced by one derived from an abstract grammar.

The ASTs in this thesis have an object-oriented foundation where the AST is a tree of objects, and links between tree nodes are references to other objects

in the tree. The objects are defined by classes in a specialization hierarchy, like in normal object-oriented programming. This allows us to make use of the typical object-oriented mechanisms such as inheritance and overriding as a basic modularization mechanism. The figure below shows a sample AST for the expression *1*2+3* built from objects in the class hierarchy described above. Computations can conveniently be modularized using the language constructs as decomposition criteria, commonly known as the Interpreter design pattern [GHJV95]. New language constructs can then be added by adding corresponding AST classes, and extending existing computations with behavior for that new construct.



3.3 The expression problem

Object-oriented ASTs allow modular extension of a language by defining new node classes for new language constructs. A class can then implement the desired computations for the language construct in a modular fashion. However, adding a new operation involves modifying every class in the system that should support that operation. The implementation of the new operation is thus scattered all over the system rather than being encapsulated in a single module. This is a result of the typical data-centric modularization of compilers using object-oriented ASTs. However, if we choose a more operation-centered modularization strategy, e.g., the Visitor design pattern [GHJV95], it is easy to add new computations, but hard to add new language elements.

It turns out that this kind of extensibility problem is fundamental to recursive data structures. While it is usually straightforward to add either new data types or new operations in a modular fashion it is hard to do both simultaneously. While the issue has been known for many years [Rey75,Coo91], the addition of parametric polymorphism to mainstream languages such as Java and C# has shown renewed interest in the problem. The problem is commonly known as the *expression problem* (named by Phil Wadler in [Wad98]) when requiring a solution to the following requirements:

– Extensibility in both dimensions, i.e., new data elements and new operations.

– Strong static type safety
– No modifications to existing source code
– Separate compilation

Proposed solutions for mainstream languages usually make clever use of generics [Tor04]. While solving the problem, a drawback is the relatively extensive and complex programming protocol that the developer has to follow. Other approaches rely on extended type systems or add additional language constructs to the language [OZ05,ZO01a,Ern04,NCM04,Bru03]. These extensions often use language constructs and type systems inspired by virtual classes [MMP89].

There are also design pattern based solutions that can be implemented without added language support but which give up the static type safety requirement. The traditional design pattern Visitor [GHJV95] is extended to support the addition of new node types. [KFF98,PJ98,NCM03].

All of the above examples require advance planning from the developer to allow for modular extensions. An alternative is provided by *open classes* [CLCM00] and *inter-type declarations* [KHH+01] which allow the developer to add new methods and fields to an existing class hierarchy in a modular fashion. Open classes support separate compilation by requiring the caller to explicitly import added methods by naming the enclosing compilation unit [CLCM00]. Inter-type declarations do not support separate compilation but make the added methods visible throughout the system by default. Our approach to the expression problem is very similar to inter-type declarations where we trade separate compilation for a convenient programming model without the need for additional programming protocols and extended type systems. However, it is worth noticing that the ReRAGs formalism can be combined with any object-oriented (data-centric) solution to the expression problem.

3.4 Aspect Oriented Software Development

Aspect oriented software development (AOSD) deals with modularization and separation of various concerns for features that cross-cut the main class hierarchy. Most programming languages, as discussed in the context of the expression problem, suffer from a limitation often called the *tyranny of the dominant decomposition*. This means that the program can only be modularized in one way at the time, but that there are concerns that do not align well with that modularization and therefore end up scattered across many modules and become tangled with other concerns. The expression problem is a typical example of this problem where, for instance, a computation cross-cuts the dominant data-centric modularization. AOSD deals with languages, tools, and techniques supporting alternative modularization concepts that allow state and behavior to

be extracted from the main class hierarchy into modules that form a separate type hierarchy [KHH+01,TOHJ99,BA01,Lie96]. A computation, such as evaluating an expression, can thus be separated into a single module instead of being scattered over multiple modules (classes).

Since the main challenges that AOSD addresses are very similar to the challenges in this thesis we relate our approach to the AspectJ language which is the most commonly used aspect-oriented language in the AOSD community. Aspects support static modularization features in the form of inter-type declarations but also a dynamic model based on well-defined points in the execution of a program. A *join-point* is such a location in the program flow and *pointcuts* are sets of join-points that can be defined using a dedicated pointcut language, querying for instance the class hierarchy, control-flow graph, and more primitive pointcuts. Cross-cutting behavior is implemented as *advice*, which is a piece of code that is executed at a set of join-points picked out using a pointcut. There are several different kinds of advice, e.g., *before advice*, *after advice*, and *around advice*. Before advice runs as a join point is reached, before the program proceeds with the join point. After advice runs after the join-point has been executed. Around advice is a combination of both before and after advice but is also in control whether the program proceeds with the join-point or not.

ReRAGs use declarative attributes defined by equations. Since the evaluation model is declarative it does not make sense to talk about join-points at the attribute level. However, individual equations may be implemented using imperative code, as long as the code is free from external side-effects. It may therefore be useful for an extension to use pointcuts to refine individual equations in the base language. However, our experience indicates that the declarative model encourages fine grained attributes that solve small dedicated problems, reducing the need for refining the code in an equation. Forthermore, we do use a language construct for refining an entire equation, similar to an around advice, at a single join-point.

3.5 Declarative computations with demand evaluation

While general-purpose object-oriented programming is imperative, many subproblems can be expressed at a higher level using declarative constructs. In an imperative language the computable relationships are expressed in terms of sequences of operations. Declarative programs, on the other hand, are made up of sets of equations describing the relations that specify what is to be computed. Declarative constructs only express what to be computed and not in which order it is to be computed. The actual order of the computations is then decided automatically.

Compilers contain context-sensitive computations such as name analysis and type checking where there are complex dependences between the various analyses. For example, in object-oriented languages, name binding and type analysis are mutually dependent: Consider a qualified access `a.b`. To do name binding of `b` we need to first do type analysis of `a`, to determine which class to query for visible fields. To do type analysis of `a` we need to do name binding of `a`, to detect its declared type. Name binding and type analysis are thus mutually dependent, and these computations need to be carefully scheduled if implemented imperatively. Extensible compilers add complexity to this scheduling since new language constructs and analyses may introduce additional dependences between existing constructs. It is worth noticing that the expression problem discussed earlier only deals with the modularization of computations and not with dependences between the various computations. In imperative programming, the scheduling of the computations is usually implemented manually, as passes over the AST. In contrast, ReRAGs are based on declarative executable specifications, allowing the scheduling of combined computations to be automatically derived. Modular programming is about dividing a problem into sub-problems, solving each sub-problem, and then combining the individual results into a complete solution. Declarative programming allows that combination to be done automatically rather than programmed explicitly by the developer. The modularization is thus enhanced by the means in which we can combine the solution from the individual sub-problems.

Modularity can be further enhanced by combining declarative programming with *demand evaluation*. Demand evaluation attempts to delay a computation until the result of that computation is known to be needed. If a sub-problem is not needed to solve the full problem it will not be computed, reducing the overall computation cost. This allows us to specify all facts about a problem and then selectively solve only the needed sub-problems. Demand evaluation makes this selection process automatic and we need not be concerned with excessive computation cost. To exemplify, Java requires variables that are declared final to be assigned once only. Demand-driven evaluation allows us to define the "assigned once" property for all variable assignments, but compute it only when the assigned variable is actually final.

3.6 Attribute Grammars

An attribute grammar (AG) supports declarative computations by decorating an AST with attributes that are defined by equations. An evaluator automatically solves the equation system for any AST following the abstract grammar. Typical applications include the program analysis that is part of the front end of a compiler, e.g., name binding and type analysis. The basic idea was introduced by [Knu68] and a large amount of research has been done in this area.

Examples of influential systems include the Eli system by Kastens and Waite [KPJ98] and the incremental Synthesizer Generator system by Teitelbaum and Reps [RT84].

Synthesized and inherited attributes Attribute grammars attach attributes to node types and define their values using equations. An attribute is either *synthesized* or *inherited* depending on if it is used for propagating information upwards or downwards in the AST. Attribute grammars blend nicely with object-oriented languages when using an object-oriented notation for attribute grammars [Hed89]. Synthesized attributes are very similar to virtual methods in this notation: Attribute declarations correspond to virtual methods in abstract classes that are overridden by equations in subclasses. Consider the following code snippet that defines an attribute called `pp()` for all `Expressions`, defining a pretty printed string for the `Expression` subtree.

```
// all expressions can be pretty printed
syn String Expression.pp()
// literals are integer values
eq Literal.pp() = value();
// concatenate left operand, operator, right operand
eq BinaryExpression.pp() = left().pp()+op()+right().pp();

// define the operator for binary expressions
syn BinaryExpression.op();
eq AddExpression.op() = "+";
eq SubExpression.op() = "-";
```

A notable difference between synthesized attributes and virtual methods is that attributes may not contain side-effects. This allows the JastAdd system to translate the attributes to virtual methods that cache the individual results for efficient evaluation. Each attribute is then only computed once, even if its value is demanded several times.

Inherited attributes are used for passing information downwards in an AST, giving child nodes information about their context. Note that the term "inherited" attribute has different meanings in attribute grammars and object-orientation. The use of inherited attributes decouples an AST node from its parent: the AST node does not need to know which parent it has. All the information it needs is in the inherited attributes whose values are defined by the parent. Allowing the parent to define the attribute for all its descendants, rather than the direct child only, further decouples an AST node from its descendants [KW94]. This is useful to model any nested structures. Consider the snippet below that defines the name of the enclosing class for all expressions within that class. The equation for `className()` is defined by `ClassDecl` and is valid

for all inherited attributes named `className()` in the subtree that is rooted in a body declaration to that `ClassDecl`.

```
// all expressions are enclosed in a class
inh String Expr.className();
// the class defines the name for the entire subtree
// that is rooted in a body declaration
eq ClassDecl.getBodyDecl().className() = name();
```

Extensions to attribute grammars There are numerous extensions to AGs that enhance modularity [KW94,FMY92,GG84,dMJW00,DC90] and genericity [SS99,dMBS00]. Other extensions include references to remote nodes [Hed94,Hed00,Boy96,PH97], to dynamically change the structure of the AST [VSK89,Sar99,EH04], and support for circular dependences between equations [Far86,MH03]. The most relevant formalisms in the context of our work are concerned with references, circularities, and dynamic creation of trees that can be attributed.

Reference Attributed Grammars (RAGs) are AGs augmented with the possibility for attributes to be *references* that refer to arbitrary syntax nodes. This allows for very natural specification of many program analysis problems, like name and type analysis for object-oriented languages (something that becomes very convoluted when using plain AGs). For example, the class hierarchy in an OO program can be represented by direct links between the class declaration AST nodes. The basic idea has emerged independently from a few researchers, described in [Hed94,Hed00,Boy96,PH97].

References to remote tree nodes can also make use of object-oriented abstraction mechanisms in that queries can be delegated to remote nodes through their abstract interfaces without knowing implementation details. This has been used successfully in name analysis to delegate lookup of members to the declaring class [EH06a]. References also reduce coupling for remote dependences where the computation of the remote location and the desired property of that location can be separated. For instance, the reference to a super-class is decoupled from the lookup of inherited members in that class. This separation can also remove circularities compared to attributes that hold an entire environment [MH03].

The original definition of attribute grammars does not allow *circular dependences* between attribute equations. However, mutually dependent equations are very useful to specify many analyses, e.g., data-flow analysis and code optimization. Circular Attribute Grammars (CAGs) [Far86] allow circular dependences between equations and compute well defined values when all attributes in a cycle have a fix-point that can be computed with a finite number of iterations. Circular Reference Attributed Grammars (CRAGs) combine both CAGs

and RAGs in one formalism with enhanced support for both non-local dependences and fix-point computations [MH03]. CRAGs have been implemented in the JastAdd tool.

The AST produced by a parser is seldom ideal for all analyses during compilation. There is therefore a need to *dynamically change the tree structure* after the AST is partially attributed. Higher Order Attribute Grammars (HAGs) [VSK89,Sar99] allow attributes to be new ASTs, with applications in, e.g., macro processing. ReRAGs provide an alternative approach where a partially attributed subtree may be rewritten allowing for iterative refinement of the AST. This is further discussed in the context of transformation systems below.

Attribute evaluators Since attribute grammars are declarative they rely on an attribute evaluator to derive a suitable computation order. Attribute evaluators can be divided into two categories: static and dynamic evaluators. *Static evaluators* analyze an attribute grammar and compute an evaluation order from the dependences between attribute equations in the grammar. This evaluator can then be used for any tree of that grammar. Static evaluators approximate the actual dependences in an AST, and are therefore less powerful than dynamic evaluators. In particular, it is difficult to use static evaluation for AGs with reference attributes, although there is some work in that direction [Boy05]. A survey of static evaluation algorithms, both batch oriented and incremental ones, is available in [Alb91].

Dynamic evaluators compute a dependency graph, at run-time, for the equations in the AST to be attributed. This graph is then sorted in topological order and the result is used as the evaluation order. This sort can either be done *explicitly* in a separate pass (before attribute evaluation) or *implicitly* using the following translation scheme: Each access to an attribute value is replaced with a function call that computes the appropriate semantic function (equation right-hand side) for that value. This simple scheme results in an evaluation in topological order [Mad80,Jal83,Jou84]. To increase efficiency, the results of individual attribute computations are usually *cached* for later accesses. This implicit scheme is more powerful than computing the dependences before evaluation, because it allows the dependences themselves to depend on the attribute values, and not only on the AST structure.

The JastAdd system uses this dynamic implicit evaluation scheme, combined with attribute caching. The algorithm supports RAGs and a convenient implementation using object-oriented programming is presented in [HM03]. Extended algorithms that support CRAGs and ReRAGs are presented in [MH03] and [EH04] respectively. Dynamic evaluators are often considered slow, but our

work shows that they can be used successfully for complex languages such as Java, processing bodies of code larger than 100kLOC.

Attribute grammars provide powerful techniques to decouple the declaration of an attribute from its implementation. The support for references allows the AST to be the only data structure in the compiler and further reduces coupling between remote nodes. Combined with inter-type declarations, enabling modularization of AST computations, they allow modular extension of computations in compilers.

3.7 Transformation systems

Tree transformation systems use rewrite rules to specify how to transform a part of an AST into another shape. An introduction to tree transformation systems is available in [Klo92] and influential systems include ASF+SDF [vdBvDH$^+$01] and Stratego [Vis01] which are both based on algebraic specifications, and TXL [Cor04] which is based on functional programming. Typical applications are code optimizations and reengineering of program source code.

Transformation systems need to take contextual computations into account when processing programs. A rewrite rule may have a condition that needs to be true for that rule to fire. Contextual information may then be used to select appropriate rules and to compute the shape of the rewritten tree. Context-sensitive data is often kept in an external database that is updated during the transformation. This requires the user to explicitly associate updates with rewrite rules or distinct passes. Contextual computations can be embedded as side-effects in rewrite rules. The application order of rewrite rules need then to take contextual dependences, which can be highly nonlocal, into account. The specification of rewrite rules and contextual dependences can be modularized using user-defined transformation strategies. Separate rules are then used to compose user-defined strategies from simpler ones using a set of combinators. A survey on strategies in program transformation systems is available in [Vis05].

ReRAGs support declarative context-sensitive conditional rewriting, combining attribute grammars with transformation technology. An important difference between ReRAGs and transformation systems is the handling of contextual information. Attribute computations and rewriting are interleaved and need not rely on manual scheduling of transformations and context-sensitive computations. The fine-grained interaction between attribute computations and rewriting is a powerful language mechanism to support extensibility through iterative refinement of the AST. Higher-order attribute grammars [VSK89,Sar99] combined with forwarding [VWMBK02] allows similar fine-grained interaction. Another important difference is how the transformation order is specified.

ReRAGs uses the dependences in contextual computations to automatically schedule the transformation order while transformation systems usually use traversal strategies to drive contextual computations. The Stratego system has a mechanism for dependent dynamic transformation rules [OV05], supporting certain context-dependent transformations, but it is not clear how this could be used for mutually dependent computations such as name binding and type checking in object-oriented languages.

3.8 Extensible Java compilers

The JastAdd Extensible Java Compiler is a full Java 1.4 compiler that has been implemented as part of this thesis work. This section compares that work to existing Java compilers from an extensibility point of view. There are quite a few Java compilers publically available, some even licensed through a generous open-source license. However, while traditional compilers may provide well-engineered APIs for adding additional analyses, e.g., the JDT model in the Eclipse Java compiler [ecl06], they are often less suited for language extensions. For instance, the ajc compiler for AspectJ [asp06] 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 extensiblity 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 front-end which has been extended successfully for numerous language features. The front-end 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 back-end 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 front-end and back-end [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 extensiblity. JaCo has later been implemented in Keris, an extension to Java that supports extensible modules with explicit refinement and specialization mechanisms [Zen04]. Both compilers are based on explicit scheduling of multiple passes.

The above compilers all rely on manual scheduling of dependences between analyses. In contrast, the JastAdd Extensible Java compiler is implemented in the declarative ReRAGs formalism, combining the language mechanisms supporting modularity and extensibility described in the previous sections. The extensible specification is only two-thirds the size of the smallest handwritten compiler. The generated compiler passes at least as many tests as production compilers and the compilation times are well within a factor of four compared to the fastest Java-based compiler.

4 Contributions

The goal of this thesis is to provide techniques and tools for high-level implementation of program-processing tools. Of particular importance is the modularization and extensibility of complex context-sensitive computations in object-oriented languages. We focus on the following three different decomposition criteria for modularization:

Separate Computations To decompose a compiler or similar tool into separate computations on the program model.

Language Extension To decompose a compiler or similar tool into an implementation of a base language and extensions to the language.

Language Specification To use the same decomposition of the compiler as is used for the informal language specification.

We notice the importance for techniques to scale to full languages as well as large sized applications to be of practical use.

This section presents the contributions of this thesis where each subsection corresponds to one of the five papers included in the thesis. The author of this thesis is the primary author of all the included papers.

4.1 Rewritable Reference Attributed Grammars

Paper I presents an object-oriented technique for rewriting abstract syntax trees in order to simplify compilation. The technique, Rewritable Reference Attributed Grammars (ReRAGs), is completely declarative and supports both rewrites and computations through attributes. ReRAGs uses several synergistic mechanisms for supporting separation of concerns: inheritance for model modularization, inter-type declarations for cross-cutting concerns, and rewrites that allow computations to be expressed on the most suitable model. The paper presents the ReRAG formalism, its evaluation algorithm, and examples of its use. We also present several typical ways to transform the AST that we discovered when implementing the JastAdd extensible Java compiler in ReRAGs. These transformations have substantially simplified the implementation as compared to having to program this by hand, or having to use a plain RAG on the initial AST constructed by the parser.

4.2 The JastAdd System

Paper II presents the JastAdd system which enables open modular specifications of extensible compiler tools and languages using the ReRAGs and CRAGs formalisms. This paper presents key language mechanisms and how they support implementation of extensible compiler tools and languages. A key design idea is to make use of declarative specification mechanisms in order to allow a high degree of decoupling between different modules, thereby supporting reuse and extensibility. We demonstrate the full power of the tool using the JastAdd extensible Java compiler, a complete Java 1.4 compiler, as a very strong case. To demonstrate extensibility both for *Separate Computations* and *Language Extension*, we have extended the compiler with new language constructs from Java 1.5 and extended the front-end with devirtualization analysis. All these extensions have been done in a completely modular way.

The current version of the JastAdd system is implemented as part of the thesis work, and builds loosely on an older version implemented by Görel Hedin and Eva Magnusson[HM03]. That system was implemented in Java and supported the CRAGs formalism only. The current system is a re-implementation in ReRAGs, that has been bootstrapped in itself, and supports both ReRAGs and CRAGs. The system has also been extended with improved support for inter-type declarations, parameterized attributes, and various convenient shorthands. The system is available at http://jastadd.cs.lth.se

4.3 The JastAdd Extensible Java Compiler

The JastAdd Extensible Java compiler is a full Java 1.4 compiler including compile-time checks and bytecode generation implemented in the ReRAGs specification formalism. Java 1.4 is a large complex language and implementing a complete compiler for it is a substantial undertaking, both because the language contains many idiosyncrasies that must be handled, and because it is an object-oriented language with many non-trivial constructs. In paper III we compare our compiler, generated from a ReRAGs specification, to other well-known Java compilers in terms of language compliance, compilation time, and implementation size. The generated compiler passes at least as many tests as production compilers such as javac [jav06], the Eclipse Java Compiler [ecl06], and jikes [jik06]. We demonstrate the *scalability* of the generated compiler by compiling a subset of the JDK of over 100.000 lines of code. The compilation time is well within a factor of four compared to the handwritten compilers implemented in Java while the specification is only two-thirds the size of the smallest handwritten compiler.

The JastAdd Extensible Java compiler has been implemented as part of the thesis work. The scanner and parser are implemented using traditional parsing technology. The lexical grammar in the Java Language Specification [GJSB00] is used to generate a scanner. The parser is generated from the context-free syntactic grammar which has been slightly modified to be LALR(1). Semantic actions are used to build the initial tree according to an AST grammar. The complete static-semantic analysis and byte-code generation is then specified using ReRAGs. The compiler is available at http://jastadd.cs.lth.se

4.4 Modular name analysis for Java using JastAdd

Name analysis for Java is challenging with its complex visibility rules involving nested scopes, inheritance, qualified access, and syntactic ambiguities. In paper IV we show how Java name analysis including ambiguities related to names of variables, fields, and packages, can be implemented in a declarative and modular manner using the JastAdd compiler construction system. The techniques illustrate the *Language Specification* decomposition criterion, where declarative attributes and context-dependent rewrites enable the implementation to be modularized in the same way as the informal Java Language Specification. The individual rules in the specification transfer directly to equations in the implementation, enabling simple traceability between the specification and the executable implementation.

4.5 Pluggable non-null types for Java

Static type systems allow for early detection of errors and enable developers to clearly document their intent. This paper demonstrates how the JastAdd extensible Java compiler can be extended with a pluggable non-null type system. Non-null types is a type-based approach to statically detect possible null-pointer violations in code. A type refinement implementation that automatically infers non-null types in legacy code is also presented. The implementation constitutes a strong case for extensibility in the JastAdd system, demonstrating how the declarative features of reference attributes and circular attributes can be taken advantage of to provide a compact modular implementation of a non-trivial type-system addition. The type-system extension is an example of *Language Extension* while the inference module is an example of *Separate Computations*. Both implementations are compact, the non-null extension being around 220 lines of code while the type refinement algorithm is less than 460 lines of code.

5 Conclusions and future work

We have presented Rewritable Reference Attributed Grammars, a high-level formalism for developing program-processing tools, e.g., compilers, consistency checkers, and reengineering tools. These tools perform similar analyses and can therefore benefit from shared infrastructure. ReRAGs support several different decomposition criteria for modularization that enable such re-use: as *separate computations* on the program model, as a base language and *language extensions*, and the same decomposition as used in a *language specification* for traceability. ReRAGs provide declarative attributes and rewriting to decouple such modules from each other and to automatically resolve complex context-sensitive dependences.

We have implemented the JastAdd extensible Java compiler to evaluate the ReRAGs formalism and to provide a platform for Java extension and source code analysis development. It is demonstrated how Java name analysis including ambiguities related to names of variables, fields, and packages, can be implemented in a declarative and modular manner while closely following the informal language specification. We also show how to extend Java with a pluggable non-null type system and also to add a separate computation that conservatively infers that property in legacy code. All extensions are made in a completely modular fashion.

The techniques scale to real languages and large applications. Our generated Java compiler passes as many tests as production use compilers during compliance testing and compiles applications larger than 100.000 lines of code.

There are several interesting ways to continue this work:

Applications We are currently implementing Java 5 as modular language extensions to the JastAdd Extensible Java compiler. Our progress so far is very promising, having implemented most parts of generics including wildcards in a completely modular fashion. The implementation uses a similar technique as described in [EH06b].

The analyses we have made so far are mostly concerned with static-semantic analysis of object-oriented languages. We believe that the combination of context-sensitive computations and rewriting can be equally useful in other areas as well. Refactorings typically depend on context-sensitive computation when restructuring the source code, and ReRAGs should therefore be a suitable specification formalism. We would also like to explore more traditional code optimization, particular by combining ReRAGs and CRAGs.

Traceability Our work on modular name analysis for Java shows that it is possible to modularize the implementation to provide traceability to a specification [EH06a]. We believe that the same techniques can be used to implement a type system that closely follows a formal proof of type soundness, or a refactoring tool with traceability to a behavior preservation proof.

Abstractions The modularization mechanisms presented in this thesis provide excellent support for extensible compiler implementation. They do, however, break encapsulation compared to traditional object-oriented programming. To improve abstraction, we therefore need new encapsulation mechanisms that allow us to selectively hide the implementation details of an analysis, when being used by other analyses, while exposing enough detail to a language extension.

We have discovered several idioms that often occur in compiler implementation when using ReRAGs. Some of these may actually be useful to evolve into language constructs. Such high-level constructs could further strengthen the abstraction mechanisms in JastAdd.

Safety We have shown how to statically detect possible null-pointer violations in Java by adding non-null types [EH06b]. Extended type systems can be used to improve the JastAdd system as well. Non-null types would, for instance, ensure that reference attributes always refer to AST nodes. Safety could be further enhanced by adding immutable types, which can be used to guarantee that code implementing attribute equations are free from side-effects.

Efficiency The evaluation algorithm in JastAdd relies heavily on caching of attribute values for efficiency [EH04]. There is a penalty, both in terms of memory consumption and execution time, when storing a value in the cache. We

believe that it is possible to statically analyze the grammar and conservatively detect attributes that are rarely accessed, and thus do not benefit from caching. A selective caching strategy based on this analysis would lead to increased efficiency.

A related optimization concerns flushing of attribute caches. The current implementation never discards cached values, which can lead to high memory consumption. However, it is safe to discard cached values, since the demand-driven evaluator will re-compute the value the next time it is accessed. It would thus be interesting to experiment with support for flushing caches to reduce memory footprint.

The ReRAGs evaluation algorithm disables caching of attributes during rewriting, since a transformation may cause attribute values to change. Incremental evaluation of attributes would allow more attributes to be cached, and re-evaluated only when needed by an incremental update. This could lead to improved performance but also to better support for interactive systems based on ReRAGs. However, incremental evaluation of attributes is quite tricky when supporting references [Boy05] and rewriting would probably make it even harder.

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.

[Alb91] Henk Alblas. Attribute evaluation methods. In *Proceedings on Attribute Grammars, Applications and Systems*, pages 48–113, London, UK, 1991. Springer.

[asp06] ajc in the AspectJ project, 1.5.0, 2006. http://www.eclipse.org/aspectj/.

[BA01] Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, 2001.

[Boy96] John Tang Boyland. *Descriptional Composition of Compiler Components*. PhD thesis, University of California, Berkeley, September 1996. Available as technical report UCB//CSD-96-916.

[Boy05] John Tang Boyland. Remote attribute grammars. *J. ACM*, 52(4):627–687, 2005.

[BP01] Jonthan Bachrach and Keith Playford. The Java syntactic extender (JSE). In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 31–42, New York, NY, USA, 2001. ACM Press.

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

[Bru03] Kim B. Bruce. Some challenging typing issues in object-oriented languages. *Electr. Notes Theor. Comput. Sci.*, 82(7), 2003.

[BV04] Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Douglas C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000. ACM Press.

[Coo91] William R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/-Workshop, Noordwijkerhout, The Netherlands, May 28 - June 1, 1990, Proceedings*, volume 489 of *LNCS*, pages 151–178. Springer, 1991.

[Cor04] James R. Cordy. Txl: A language for programming language tools and applications. In *Proceedings of the 4th Workshop on Language Descriptions, Tools, and Applications (LDTA'04) at ETAPS 2004*, 2004.

[DC90] G. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *Comput. J.*, 33(2):164–172, 1990.

[Dij82] Edsger W. Dijkstra. *On the role of scientific thought, in Selected Writings on Computing: A Personal Perspective*. Springer, Secaucus, NJ, USA, 1982.

[dMBS00] Oege de Moor, Kevin Backhouse, and S. Doaitse Swierstra. First-class attribute grammars. *Informatica (Slovenia)*, 24(3), 2000.

[dMJW00] Oege de Moor, Simon L. Peyton Jones, and Eric Van Wyk. Aspect-oriented compilers. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, pages 121–133, London, UK, 2000. Springer.

[ecl06] Eclipse java compiler, eclipse java development tools 3.1.2, 2006. http://download.eclipse.org/eclipse/downloads/drops/R-3.1.2-200601181600/

[EH04] Torbjörn Ekman and Görel Hedin. Rewritable reference attributed grammars. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, volume 3086 of *LNCS*, pages 144–169. Springer, 2004.

[EH06a] Torbjörn Ekman and Görel Hedin. Modular name analysis for Java using JastAdd. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering, Braga, Portugal, July 4–8, 2005*, LNCS. Springer, 2006. To appear.

[EH06b] Torbjörn Ekman and Görel Hedin. Pluggable non-null types for java. Technical report, 2006. Unpublished manuscript, http://jastadd.cs.lth.se.

[Ern04] Erik Ernst. The expression problem, scandinavian style. In Philippe Lahire and et al., editors, *Proceedings of MASPEGHI 2004*, ISRN I3S/RR-2004-15-FR, Oslo, Norway, June 2004. Laboratoire I3S, Sophia Antipolis.

[Far86] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the SIGPLAN symposium on Compiler contruction*, pages 85–98. ACM Press, 1986.

[FMY92] Rodney Farrow, Thomas J. Marlowe, and Daniel M. Yellin. Composable attribute grammars: Support for modularity in translator design and implementation. In *19th ACM Symp. on Principles of Programming Languages*, pages 223–234, Albuquerque, NM, January 1992. ACM press.

[GG84] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 157–170, New York, NY, USA, 1984. ACM Press.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

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

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Third Edition*. Addison-Wesley, Boston, Mass., 2005.

[Hed89] Görel Hedin. An object-oriented notation for attribute grammars. In *the 3rd European Conference on Object-Oriented Programming (ECOOP'89)*, BCS Workshop Series, pages 329–345. Cambridge University Press, July 1989.

[Hed94] Görel Hedin. An overview of door attribute grammars. In Peter A. Fritzson, editor, *5th Int. Conf. on Compiler Construction (CC' 94)*, volume 786 of *LNCS*, pages 31–51, Edinburgh, April 1994.

[Hed00] Görel Hedin. Reference Attributed Grammars. *Informatica (Slovenia)*, 24(3), 2000.

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

[Jal83] Fahimeh Jalili. A general linear-time evaluator for attribute grammars. *SIGPLAN Not.*, 18(9):35–44, 1983.

[jav06] javac in java 2 platform, standard edition 5.0, 2006. http://java.sun.com/j2se/1.5/.

[jik06] Jikes high performance java compiler, 1.22, 2006. http://jikes.sourceforge.net/.

[Jou84] Martin Jourdan. An optimal-time recursive evaluator for attribute grammars. In *Proceedings of the 6th Colloquium on International Symposium on Programming*, pages 167–178, London, UK, 1984. Springer.

[KFF98] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 91–113, London, UK, 1998. Springer.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *LNCS*, 2072:327–355, 2001.

[Klo92] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon*, volume 2. 1992.

[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).

[KPJ98] Uwe Kastens, Peter Pfahler, and Matthias Jung. The Eli System. In Kai Koskimies, editor, *Compiler Construction CC'98*, volume 1383 of *LNCS*, Portugal, April 1998. Springer.

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

[Lie96] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[Mad80] Ole Lehrmann Madsen. On defining semantics by means of extended attribute grammars. In *Semantics-Directed Compiler Generation, Proceedings of a Workshop*, pages 259–299, London, UK, 1980. Springer.

[McC64] John McCarthy. A formal description of a subset of ALGOL. In T. B. Steele, Jr, editor, *Formal Language Description Languages for Computer Programming, Proceedings of an IFIP Working Conference*, pages 1–12. Springer, 1964.

[MH03] Eva Magnusson and Görel Hedin. Circular reference attributed grammars - their evaluation and applications. *Electronic Notes in Theoretical Computer Science*, 82(3), 2003.

[MMP89] O. L. Madsen and B. Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proc. of the OOPSLA-89: Conference on Object-Oriented Programming: Systems*, pages 397–406, Languages and Applications, New Orleans, LA, 1989.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In Görel Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2622 of *LNCS*, pages 138–152. Springer, 2003.

[NCM04] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 99–115, New York, NY, USA, 2004. ACM Press.

[OV05] Karina Olmos and Eelco Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In Rastislav Bodík, editor, *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3443 of *LNCS*, pages 204–220. Springer, 2005.

[OZ05] Martin Odersky and Matthias Zenger. Independently extensible solutions to the expression problem. In *Proc. FOOL 12*, January 2005. `http://homepages.inf.ed.ac.uk/wadler/fool`.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[PH97] A. Poetzsch-Heffter. Prototyping realistic programming languages based on formal specifications. *Acta Informatica*, 34(10):737–772, 1997.

[PJ98] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *COMPSAC '98: Proceedings of the 22nd International Computer Software and Applications Conference*, pages 9–15, Washington, DC, USA, 1998. IEEE Computer Society.

[Rey75] John C. Reynolds. User-defined types and procedural data as complementary approaches to data abstractions. In *S. A. Schuman, editor, New Directions in Algorithmic Languages*. IFIP Working Group 2.1 on Algol, INRIA, 1975. Reprinted in: D. Gries, editor, "Programming-Methodology", Springer, 1978, and in C. A. Gunter and J. C. Mitchell, editors, "Theoretical Aspects of Object-Oriented Programming", MIT Press, 1994.

[RT84] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments*, pages 42–48. ACM press, Pittsburgh, PA, April 1984.

[Sar99] Joao Saraiva. *Purely functional implementation of attribute grammars*. PhD thesis, Utrecht University, The Netherlands, 1999.

[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.

[SS99] Joao Saraiva and Doaitse Swierstra. Generic Attribute Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 185–204, Amsterdam, The Netherlands, 1999. INRIA rocquencourt.

[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, London, UK, 2000. Springer.

[TOHJ99] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

[Tor04] Mads Torgersen. The expression problem revisited. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, volume 3086 of *LNCS*, pages 123–143. Springer, 2004.

[vdBvDH+01] Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 365–370, London, UK, 2001. Springer.

[Vis97] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[Vis01] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Proceedings of Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *LNCS*, pages 357–361. Springer, 2001.

[Vis05] Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. Special issue on Reduction Strategies in Rewriting and Programming.

[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 the SIGPLAN '89 Programming language design and implementation*. ACM Press, 1989.

[VWMBK02] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proceedings of Compiler Construction Conference 2002*, volume 2304 of *LNCS*, pages 128–142. Springer, 2002.

[Wad98] Phil Wadler. The expression problem. Posted on the Java Genericity mailing list, 1998.

[Zen04] Matthias Zenger. *Programming Language Abstractions for Extensible Software Components*. Ph.D. thesis, Department of Computer Science, EPFL, Lausanne, 2004.

[ZO01a] Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 241–252, New York, NY, USA, 2001. ACM Press.

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

Paper I

Rewritable Reference Attributed Grammars

Torbjörn Ekman and Görel Hedin

Department of Computer Science, Lund University, Sweden
(torbjorn|gorel)@cs.lth.se

Rewritable Reference Attributed Grammars

Torbjörn Ekman and Görel Hedin

Department of Computer Science, Lund University, Sweden
(torbjorn|gorel)@cs.lth.se

Abstract This paper presents an object-oriented technique for rewriting abstract syntax trees in order to simplify compilation. The technique, Rewritable Reference Attributed Grammars (ReRAGs), is completely declarative and supports both rewrites and computations through attributes. We have implemented ReRAGs in our aspect-oriented compiler compiler tool JastAdd II. Our largest application is a complete static-semantic analyzer for Java 1.4. ReRAGs uses three synergistic mechanisms for supporting separation of concerns: inheritance for model modularization, aspects for cross-cutting concerns, and rewrites that allow computations to be expressed on the most suitable model. This allows compilers to be written in a high-level declarative and modular fashion, supporting language extensibility as well as reuse of modules for different compiler-related tools. We present the ReRAG formalism, its evaluation algorithm, and examples of its use. Initial measurements using a subset of the Java class library as our benchmarks indicate that our generated compiler is only a few times slower than the standard compiler, javac, in J2SE 1.4.2 SDK. This shows that ReRAGs are already useful for large-scale practical applications, despite that optimization has not been our primary concern so far.

1 Introduction

Reference Attributed Grammars (RAGs) have proven useful in describing and implementing static-semantic checking of object-oriented languages [Hed00]. These grammars make use of *reference attributes* to capture non-local tree dependences like variable decl-use, superclass-subclass, etc., in a natural, yet declarative, way.

The RAG formalism is itself object-oriented, viewing the grammar as a class hierarchy and the abstract syntax tree (AST) nodes as instances of these classes. Behavior common to a group of language constructs can be specified in superclasses, and can be further specialized or overridden for specific constructs in the corresponding subclasses.

In plain RAGs, the complete AST is built prior to attribute evaluation. While this works well for most language constructs, there are several cases

where the most appropriate tree structure can be decided only *after* evaluation of some of the attributes. I.e., the context-free syntax is not sufficient for building the desired tree, but contextual information is needed as well. By providing means for rewriting the AST based on a partial attribution, the specification of the remaining attribution can be expressed in a simpler and more natural way.

This paper presents ReRAGs, Rewritable Reference Attributed Grammars, which extend RAGs with the capability to rewrite the AST dynamically, during attribute evaluation, yet specified in a declarative way. ReRAGs form a conditional rewrite system where conditions and rewrite rules may use contextual information through the use of attributes. We have implemented a static-semantics analyzer for Java using this technique. Based on this experience we exemplify typical cases where rewriting the AST is useful in practice.

ReRAGs are closely related to Higher-ordered Attribute Grammars (HAGs) [VSK89], [Sar99] and to the technique of forwarding in HAGs [VWMBK02]. A major difference lies in the object-oriented basis of ReRAGs, where reference attributes are kept as explicit links in the tree and subtrees are rewritten in place. HAGs, in contrast, have a functional programming basis, viewing the AST as well as its attributes as structured values without identity.

ReRAGs also have similarities to tree transformation systems like Stratego [Vis01b], ASF+SDF [vdBea01], and TXL[Cor04], but improves data acquisition support through the use of RAGs instead of embedding contextual data in rewrite rules. Rewrite application strategies differ in that ReRAGs only support the above described declarative approach while the above mentioned systems support user defined strategies. In Stratego and AST+SDF the rewrite application strategy is specified through explicit traversal strategies and in TXL the rewrite application order is implicitly defined as part of the functional decomposition of the transformation ruleset.

The plain RAG evaluation scheme is demand driven, evaluating an attribute only when its value is read. The ReRAG evaluation scheme extends this basic approach by rewriting parts of the AST as needed during the evaluation. We have designed different caching strategies to achieve performance optimization and evaluated the approach using a subset of the J2SDK 1.4.2 class library as our benchmark suite.

ReRAGs are implemented in our tool JastAdd II, a successor to our previous tool JastAdd that supported plain RAGs [HM03]. Several grammars have been developed for JastAdd II, the largest one being our Java grammar that implements static-semantics checking as specified in the Java Language Specification [GJSB00].

In addition to RAG/ReRAG support, the JastAdd systems support static aspect-oriented specification and integration with imperative Java code. Specifications are aspect-oriented in that sets of attributes and equations concerning

a particular aspect, such as name analysis, type checking, code generation, etc., can be specified in modules separate from the AST classes. This is similar to the static introduction feature of AspectJ [KHH+01] where fields, methods, and interface implementation clauses may be specified in modules separate from the original classes.

Integration with imperative Java code is achieved by simply allowing ordinary Java code to read attribute values. This is useful for many problems that are more readily formulated imperatively than declaratively. For example, a code emission module may be written as ordinary Java code that reads attribute values from the name and type analysis in order to emit the appropriate code. These modules are also specified as static introduction-like aspects that add declarations to the existing AST classes. The ReRAG examples given in this paper are taken from our experience with the Java grammar and utilize the separation of concerns given by the aspect-oriented formulation, as well as the possibility to integrate declarative and imperative modules.

The rest of this paper is structured as follows. Section 2 introduces some typical examples of when AST rewriting is useful. Section 3 gives background information on RAGs and ASTs. Section 4 introduces ReRAG rewriting rules. Section 5 discusses how ReRAGs are evaluated. Section 6 describes the algorithms implemented in JastAdd II. Section 7 discusses ReRAGs from both an application and a performance perspective. Section 8 compares with related work, and Section 9 concludes the paper.

2 Typical examples of AST rewriting

From our experience with writing a static-semantics analyzer for Java, we have found many cases where it is useful to rewrite parts of the AST in order to simplify the compiler implementation. Below, we exemplify three typical situations.

2.1 Semantic specialization

In many cases the same context-free syntax will be used for language constructs that carry different meaning depending on context. One example is names in Java, like `a.b`, `c.d`, `a.b.c`, etc. These names all have the same general syntactic form, but can be resolved to a range of different things, e.g., variables, types, or packages, depending on in what context they occur. During name resolution we might find out that `a` is a class and subsequently that `b` is a static field. From a context-free grammar we can only build generic `Name` nodes that must capture all cases. The attribution rules need to handle all these cases and therefore become complex. To avoid this complexity, we would like to do *semantic specialization*. I.e., we would like to replace the general `Name` nodes with

more specialized nodes, like `ClassName` and `FieldName`, as shown in Figure 1. Other computations, like type checking, optimization, and code generation, can benefit from this rewrite by specifying different behavior (attributes, equations, fields and methods) in the different specialized classes, rather than having to deal with all the cases in the general `Name` class.

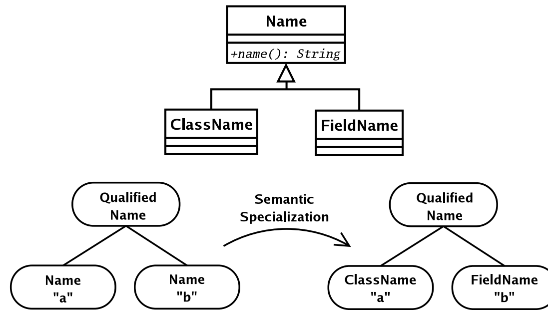


Figure 1. Semantic specialization of name references.

2.2 Make implicit behavior explicit

A language construct sometimes has *implicit behavior* that does not need to be written out by the programmer explicitly. An example is the implicit constructors of Java classes. If a class in Java has no constructors, this corresponds to an implicit constructor taking no arguments. The implicit behavior can be made explicit by rewriting the AST, see Figure 2. This simplifies other computations, like code generation, which do not have to take the special implicit cases into account.

2.3 Eliminate shorthands

Some language constructs are shorthands for specific combinations of other, more basic, constructs. For example, string concatenation in Java can be written using the binary addition operator (e.g., `a+b`), but is actually implemented as an invocation of the `concat` method in the `String` class (e.g., `a.concat(b)`). The AST can be rewritten to eliminate such shorthands, as shown in Figure 3. The AST now reflects the semantics rather than the concrete syntax, which simplifies other computations like optimizations and code generation.

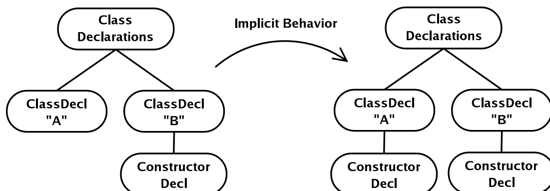


Figure 2. The implicit constructor in class “A” is made explicit.

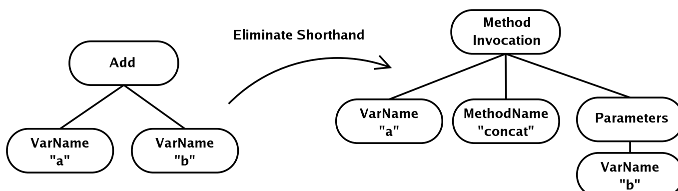


Figure 3. Eliminate shorthand and reflect the semantic meaning instead.

3 Background

3.1 AGs and RAGs

ReRAGs are based on Reference Attributed Grammars (RAGs) which is an object-oriented extension to Attribute Grammars (AGs) [Knu68]. In plain AGs each node in the AST has a number of *attributes*, each defined by an *equation*. The right-hand side of the equation is an expression over other attribute values and defines the value of the left-hand side attribute. In a consistently attributed tree, all equations hold, i.e., each attribute has the same value as the right-hand side expression of its defining equation.

Attributes can be *synthesized* or *inherited*. The equation for a synthesized attribute resides in the node itself, whereas for an inherited attribute, the equation resides in the parent node. From an OO perspective we may think of attributes as fields and of equations as methods for computing the fields. However, they need not necessarily be implemented that way. Note that the term *inherited attribute* refers to an attribute defined in the parent node, and is thus a concept unrelated to the inheritance of OO languages. In this article we will use the term *inherited attribute* in its AG meaning.

Inherited attributes are used for propagating information downwards in the tree (e.g., propagating information about declarations down to use sites) whereas synthesized attributes can be accessed from the parent and used for

propagating information upwards in the tree (e.g. propagating type information up from an operand to its enclosing expression).

RAGs extend AGs by allowing attributes to have reference values, i.e., they may be object references to AST nodes. AGs, in contrast, only allow attributes to have primitive or structured algebraic values. This extension allows very simple and natural specifications, e.g., connecting a use of a variable directly to its declaration, or a class directly to its superclass. Plain AGs connect only through the AST hierarchy, which is very limiting.

3.2 The AST class hierarchy

The nodes in an AST are viewed as instances of Java classes arranged in a subtype hierarchy. An AST class correponds to a nonterminal or a production (or a combination thereof) and may define a number of children and their declared types, corresponding to a production right-hand side. In an actual AST, each node must be *type consistent* with its parent according to the normal type-checking rules of Java. I.e., the node must be an instance of a class that is the same or a subtype of the corresponding type declared in the parent. Shorthands for lists, optionals, and lexical items are also provided. An example definition of some AST classes in a Java-like syntax is shown below.

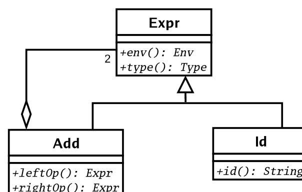
```
// Expr corresponds to a nonterminal
ast Expr;
// Add corresponds to an Expr production
ast Add extends Expr ::= Expr leftOp, Expr rightOp;
// Id corresponds to an Expr production, id is a token
ast Id extends Expr ::= <String id>;
```

Aspects can be specified that define attributes, equations, and ordinary Java methods of the AST classes. An example is the following aspect for very simple type-checking.

```
// Declaration of an inherited attribute env of Expr nodes
inh Env Expr.env();
// Declaration of a synthesized attribute type of Expr
// nodes and its default equation
syn Type Expr.type() = TypeSystem.UNKNOWN;
// Overriding the default equation for Add nodes
eq Add.type() = TypeSystem.INT;
// Overriding the default equation for Id nodes
eq Id.type() = env().lookup(id()).type();
```

The corresponding Java API is shown in the following UML diagram. It includes methods for accessing child nodes like `leftOp` and `rightOp`, tokens

like id and user-defined attributes like env and type. This API can be used freely in the right-hand sides of equations, as well as by ordinary Java code.



4 Rewrite rules

ReRAGs extend RAGs by allowing rewrite rules to be written that automatically and transparently rewrite nodes. The rewriting of a node is triggered by the first access to it. Such an access could occur either in an equation in the parent node, or in some imperative code traversing the AST. In either case, the access will be captured and a reference to the final rewritten tree will be the result of the access. This way, the rewriting process is transparent to any code accessing the AST. The first access to the node will always go via the reference to it in the parent node. Subsequent accesses may go via reference attributes that refer directly to the node, but at this point, the node will already be rewritten to its final form.

A rewrite step is specified by a rewrite rule that defines the conditions when the rewrite is applicable, as well as the resulting tree. For a given node, there may be several rewrite rules that apply, which are then applied in a certain order. It may also be the case that after the application of one rewrite rule, more rewrite rules become applicable. This allows complex rewrites to be broken down into a series of simple small rewrite steps.

A rewrite rule for nodes of class *N* has the following general form:

```
rewrite N  {
  when (cond)
  to R result;
}
```

This specifies that a node of type *N* may be replaced by another node of type *R* as specified in the result expression *result*. The rule is applicable if the (optional) boolean condition *cond* holds and will be applied if there are no other applicable rewrite rules of higher priority (priorites will be discussed later). Furthermore, all rewrite rules must be type consistent in that the replacement will result in a type consistent AST regardless of the context of the node, as will be discussed in Section 4.2. In a consistently attributed tree, all equations hold and all rewrite conditions are false.

4.1 A simple example

As an example, consider replacing an Add node with a StringAdd node in case both operands are strings [1]. This can be done as follows.

```
ast StringAdd extends Expr ::= Expr leftOp, Expr rightOp;
rewrite Add {
   when (childType().equals(TypeSystem.STRING))
   to StringAdd new StringAdd(leftOp(), rightOp());
}
syn Type Add.childType() = ...;
```

Note that in the creation of the new right-hand side, the previous children leftOp() and rightOp() are used. These accesses might trigger rewrites of these nodes in turn.

Avoiding repeated applications. StringAdd nodes might have much in common with Add nodes, and an alternative way of handling this rewrite would be to define StringAdd as a subclass of Add, rather than as a sibling class. In this case, the rewrite should apply to all Add nodes, except those that are already StringAdd nodes, and can be specified as follows.

```
ast StringAdd extends Add;
rewrite Add {
   when (childType().equals(TypeSystem.STRING)
      and !(this instanceOf StringAdd))
   to StringAdd new StringAdd(leftOp(), rightOp());
}
syn Type Add.childType() = ...;
```

Note that the condition includes a type test to make sure that the rule is not applied to nodes that are already of type StringAdd. This is necessary since the rule would otherwise still be applicable after the rewrite, resulting in repeated applications of the same rule and thereby nontermination. In general, whenever the rewrite results in the same type or a subtype, it is advicable to reflect over if the condition might hold also after the rewrite and in that case if the condition should be tightened in order to avoid nontermination.

Solutions that refactor the AST class hierarchy. A third alternative solution could be to keep Add and StringAdd as sibling classes and to factor out the common parts into a superclass as follows.

[1] In Section 4.4 we will instead rewrite addition of strings as method calls.

```
ast Expr:
ast GeneralAdd extends Expr ::= Expr leftOp, Expr rightOp;
ast Add extends GeneralAdd;
ast StringAdd extends GeneralAdd;
```

This solution avoids the type test in the rewrite condition. However, it requires that the grammar writer has access to the original AST definition of `Add` so that it can be refactored.

4.2 Type consistency

As mentioned above, rules must be *type consistent*, i.e., the replacing node must always be type consistent with any possible context. This is checked statically by the JastAdd II system. Consider the rewrite rule that replaces an `Add` node by a sibling `StringAdd` node using the grammar described above. The expected child type for all possible contexts for `Add` nodes is `Expr`. Since both `Add` and `StringAdd` are subclasses of `Expr` the rule is type consistent. However, consider the addition of the following AST class.

```
ast A ::= Add:
```

In this case the rewrite rule would not be type consistent since the rewrite could result in an `A` node having a `StringAdd` node as a child although an `Add` node is expected. Similarly, in the second rewrite example in Section 4.1 where `StringAdd` is a subclass of `Add`, that rewrite rule would not be type consistent if the following classes were part of the AST grammar.

```
ast B ::= C:
ast C extends Add;
```

In this case, the rewrite rule could result in a `B` node having a `StringAdd` node as a child which would not be type consistent.

Theorem 1. *A rule* `rewrite`*A*...`to`*B*... *is type consistent if the following conditions hold: Let C be the first common superclass of A and B. Furthermore, let $\mathcal{D}$ be the set of classes that occur on the right-hand side of any production class. The rule is type consistent as long as there is no class D in $\mathcal{D}$ that is a subclass of C, i.e., $D \not< C$.*

Proof. The rewritten node will always be in a context where its declared type D is either the same as C, or a supertype thereof, i.e. $C \leq D$. The resulting node will be of a type $R \leq B$, and since $B \leq C$, then consequently $R \leq D$, i.e., the resulting tree will be type consistent. □

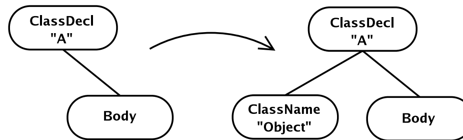
4.3 Rewriting descendent nodes

The tree resulting from a rewrite is specified as an expression which may freely access any of the current node's attributes and descendents. Imperative code is permitted, using the syntax of a Java method body that returns the resulting tree. This imperative code may reuse existing parts in the old subtree in order to build the new subtree, but may have no other externally visible side effects. This can be used to rewrite descendent nodes, returning an updated version of the node itself as the result.

As an example, consider a Java class declaration `class A { ... }`. Here, `A` is given no explicit superclass which is equivalent to giving it the superclass `Object`. In order to simplify further attribution (type checking, etc.), we would like to change the AST and insert the superclass as an explicit node. This can be done by the following rewrite rule:

```
ast ClassDecl extends Decl ::=
   <String classId>, [ TypeRef superClass ], Body body;
rewrite ClassDecl {
   when (!hasSuperClass() && !name().equals("Object"))
   to ClassDecl {
      setSuperClass(new TypeRef("Object"));
      return this;
   }
}
```

Note that the rewrite rule updates a descendent node and returns itself, as illustrated in the figure below.



As seen from the specification above, the condition for doing this rewrite is that the class has no explicit superclass already, and that it is another class than the root class `Object`. The result type is the same as the rewritten type, which means we should reflect on possible nontermination due to repeated applications of the same rule. However, it is clear that the rewrite will not be applicable a second time since the rewrite will result in a node where the condition is no longer met.

4.4 Combining rules

It is often useful to rewrite a subtree in several steps. Consider the following Java-like expression: `a+"x"`

Supposing that a is a reference to an non-null Object subclass instance, the semantic meaning of the expression is to convert a into a string, convert the string literal "x" into a string object, and to concatenate the two strings by the method concat. It can thus be seen as a shorthand for the following expression.

```
a.toString().concat(new String(new char[ ] {'x'} ))
```

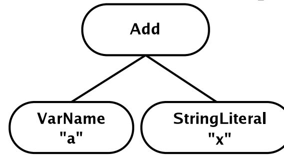
To simplify code generation we would like to eliminate the shorthand notation by rewriting the AST. This can be accomplished by a number of rewrite rules, each taking care of a single subproblem:

1. replace the right operand of an Add node by a call to toString if the left operand is a string, but the right is not
2. replace the left operand of an Add node by a call to toString if the right operand is a string, but the left is not
3. replace an Add node by a method call to concat if both operands are strings
4. replace a string literal by an expression creating a new string object

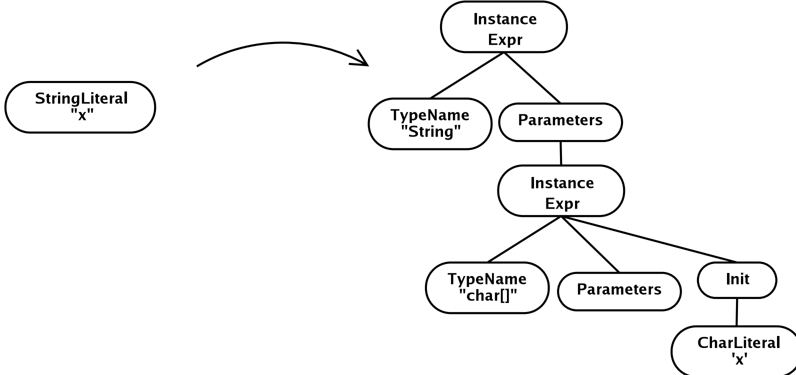
Suppose the original Add node is accessed from its parent. This will cause the AST to be rewritten in the following steps. First, it will be checked which rules are applicable for Add. This will involve accessing its left and right operands, which triggers the rewrite of these nodes in turn. In this case, the right operand will be rewritten according to rule 4. It is now found that rule 2 is applicable for Add, and it is applied, replacing the left operand by a MethodCall. This causes rule 3 to become applicable for Add, replacing it too by a MethodCall. Now, no more rules are applicable for the node and a reference is returned to the parent. Figure 4 illustrates the steps applied in the rewrite.

Rule priority. In general, it is possible that more than one rule applies to a node. Typically, this happens when there are two rewrite rules in a node, each rewriting different parts of the substructure of the node. For example, in a class declaration there may be one rewrite rule that takes care of making an implicit constructor explicit, and another rule making an implicit superclass explicit. Both these rules can be placed in the ClassDecl AST class, and may be applicable at the same time. In this particular case, the rules are *confluent*, i.e., they can be applied in any order, yielding the same resulting tree. So far, we have not found the practical use for nonconfluent rules, i.e., where the order of application matters. However, in principle they can occur, and in order to obtain a predictable result also in this case, the rules are prioritized: Rules in a subclass have priority over rules in superclasses. For rules in the same class, the lexical order is used as priority.

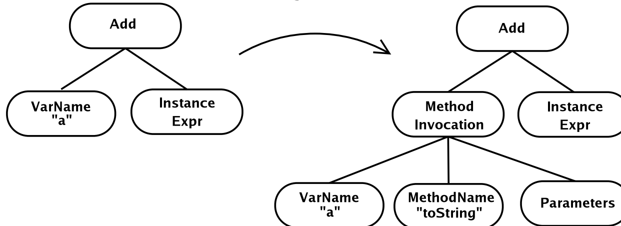
Initial AST for the `a + "x"` expression



Rule 4: Replace the `"x"` string literal by a new string instance expression `new String(new char[] {'x'})`.



Rule 2: Make the implicit Object to String type conversion explicit by adding a `"toString"` method call.



Rule 3: Replace add by a method call to `"concat"`.

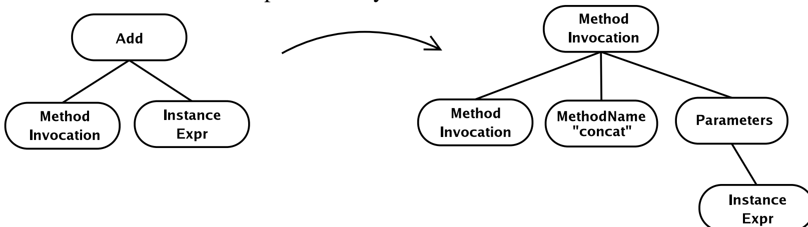


Figure 4. Combine several rules to eliminate the shorthand for String addition and literals in a Java like language.

5 ReRAG evaluation

5.1 RAG evaluation

An attribute evaluator computes the attribute values so that the tree becomes consistently attributed, i.e., all the equations hold. JastAdd uses a demand-driven evaluation mechanism for RAGs, i.e., the value of an attribute is not computed until it is read [HM03]. The implementation of this mechanism is straight-forward in an object-oriented language [Hed89]. Attributes are implemented as methods in the AST classes where they are declared. Accessing an attribute is done simply by calling the corresponding method. Also equations are translated to methods, and are called as appropriate by the attribute methods: The method implementing an inherited attribute will call an equation method in the parent node. The method implementing a synthesized attribute calls an equation method in the node itself. JastAdd checks statically that all attributes in the grammar have a defining equation, i.e., that the grammar is well-formed. For efficiency, the value of an attribute is cached in the tree the first time it is computed. All tree nodes inherit generic accessor methods to its parent and possible children through a common superclass. As a simple example, consider the following RAG fragment:

```
ast Expr;
ast Id extends Expr ::= <String id>;
inh Env Expr.env();
syn Type Expr.type();
eq Id.type() = env().lookup(id()).type();
```

This is translated to the following Java code:

```
class Expr extends ASTNode {// inherit generic node access
  Env env_value = null;           // cached attribute value
  boolean env_cached = false;      // flag true when cached
  Env env() {              // method for inherited attribute
    if(!env_cached) {
      env_value = ((HasExprSon)parent()).env_eq(this);
      env_cached = true; }
    return env_value; }
  Type type_value = null;         // cached attribute value
  boolean type_cached = false;     // flag true when cached
  Type type() {          // method for synthesized attribute
    if(!type_cached) {
      type_value = type_eq();
      type_cached = true; }
    return type_value; }
  abstract Type type_eq(); }
```

```
interface HasExprSon {
  Env env_eq(Expr son); }
class Id extends Expr {
  String id() { ... }
  Type type_eq() {          // method for equation defining
    return env().lookup(id()).type() // synthesized attr.
  } }
```

This demand-driven evaluation scheme implicitly results in topological-order evaluation (evaluation order according to the attribute dependences). See [Hed00] for more details.

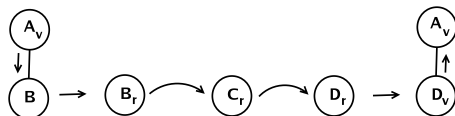
Attribute evaluation using this scheme will often follow complex tree traversal patterns, often visiting the same node multiple times in order to evaluate all the attributes that a specific attribute depends on. For example, consider the evaluation of the attribute *Id.type* above. This involves finding the declaration of the identifier, then finding the declaration of the type of the identifier, and during this process, possibly finding the declarations of classes in the superclass chain where these declarations may be located. In this process, the same block nodes and declaration nodes may well be visited several times. However, once a certain attribute is evaluated, e.g., the reference from a class to its superclass, that computation does not need to be redone since the attribute value is cached. The traversals do therefore not always follow the tree structure, but can also follow reference attributes directly, e.g., from subclass to superclass or from variable to declaration.

5.2 Basic rewrite strategy

To handle ReRAGs, the evaluator is extended to rewrite trees in addition to evaluating attributes, resulting in a consistently attributed tree where all equations hold and all rewrite conditions are false. A demand-driven rewriting strategy is used. When a tree node is visited, the node is rewritten iteratively. In each iteration, the rule conditions are evaluated in priority order, and the first applicable rule will be applied, replacing the node (or parts of the subtree rooted at the node). The next iteration is applied to the root of the new subtree. The iteration stops when none of the rules are applicable (all the conditions are false), and a reference to the resulting subtree is then returned to the visiting node. The subtree may thus be rewritten in several steps before the new subtree is returned to the visiting node. Since the rewrites are applied implicitly when visiting a node, the rewrite is transparent from a node traversal point of view.

The figure below shows how the child node *B* of *A* is accessed for the first time and iteratively rewritten into the resulting node *D* that is returned to the parent *A*. The subscript *v* indicates that a node has been visited and *r* that a

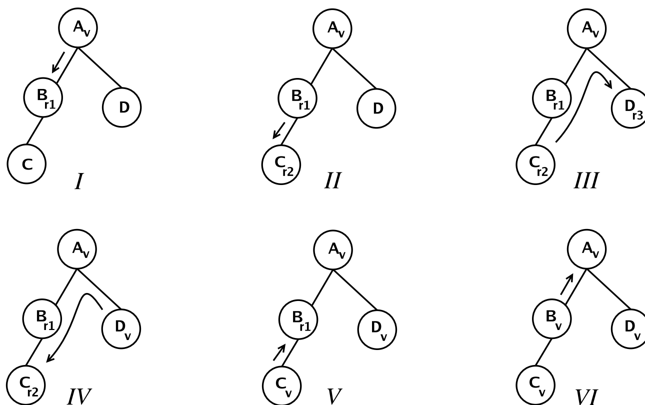
rewrite is currently being evaluated. When *B* is visited a rewrite is triggered and the node is rewritten to a *C* node that in turn is rewritten to a *D* node. No rewrite conditions for the *D* node are true, and the node is returned to the parent *A* that need not be aware of the performed rewrite.



5.3 Nested and multi-level rewrites

When evaluating a condition or a result expression in a rewrite rule, attributes may be read that trigger a visit to another node. That visit may in turn trigger a second rewrite that is executed before the first may continue its evaluation. This nesting of rewrites results in several rewrites being active at the same time. Since attributes may reference distant subtrees, the visited nodes could be anywhere in the AST, not necessarily in the subtree of the rewritten tree.

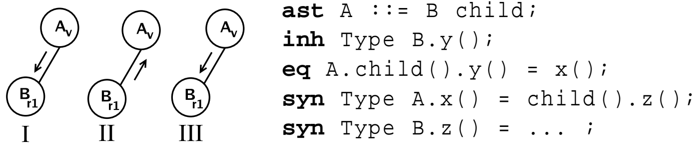
The following figure shows an example of nested rewrites. The subscript *v* indicates that a node has been visited and *r* that a rewrite is currently being evaluated. The rewrites are numbered in the order they started.



An initial rewrite, *r1*, is triggered when *A* visits its child *B* in stage *I*. A visit to *C*, that is caused by accessing a synthesized attribute during rewrite condition checking, triggers a second rewrite *r2* in stage *II*. That rewrite triggers a visit to a distant node *D* by reading an inherited attribute and initiates a third rewrite *r3* in stage *III*. When no conditions in *D* are true the result of

the inherited attribute is calculated and returned to *C* in stage *IV*. The synthesized attribute is calculated and returned to *B* in stage *V*. The resulting node *B* is finally returned to *A* in stage *VI*. Notice that the rewrites terminate in the opposite order that they were initiated.

As discussed in Section 5.1, most non-trivial attribute grammars are multi-visit in that a node may have to be visited multiple times to evaluate an attribute. A common situation is when a child node has an inherited attribute, and the equation in the parent node depends on a synthesized attribute that visits the child node again. The situation is illustrated in the figure below. *A* visits *B* and a rewrite is initiated in stage *I*. During condition evaluation the inherited attribute *y()* is read and *A* is visited to evaluate its equation in stage *II*. That equation contains the synthesized attribute *x()* that in turn depends on *z()* in *B* and a second visit is initiated in stage *III*.



Such multi-visits complicate the rewrite and attribute evaluation process somewhat. Should the second visit to a node that is being rewritten start a second rewrite? No. The attributes read in a node that is being rewritten should reflect the current tree structure. Otherwise, the definition of rewrites would be circular and evaluation would turn into an endless loop. Therefore, when visiting a node that is already being rewritten, the current node is returned and no new rewrite is triggered.

Note that attribute values that depend on nodes that are being rewritten, might have different values during the rewrite than they will have in the final tree. Therefore, such attributes will not be cached until all the nodes they depend on have reached their final form. We will return to this issue in Section 6.3.

Note also that a node may well be rewritten several times, provided that the previous rewrite has completed. This can happen if the rewrites are triggered by the rewriting of another node. For example, suppose we are rewriting a node *A*. During this process, we visit its son node *S* which is then rewritten to S'. After this rewrite of *S*, the conditions of S' are all false (the rewrite of S completes). We then complete one iteration of rewriting *A*, replacing it with a new node A'(but keeping the son S'). In the next iteration of rewriting A', it may be found that S' needs to be rewritten again since the conditions of S' may give other results after replacing *A* by A'. This will also be discussed more in Section 6.2.

6 Implementation algorithm

6.1 Basic algorithm

As discussed in Section 3.2, a Java class is generated for each node type in the AST. All classes in the class hierarchy descend from the same superclass, *ASTNode*, providing generic traversal of the AST by the generic *parent()* and *child(int index)* methods. These methods are used in the implementation of attribute and equation methods, as discussed in Section 5.1.

We have implemented our rewriting algorithm by extending the existing JastAdd RAG evaluator as an AspectJ [KHH+01] aspect. In particular, the *child* method is extended to trigger rewrites when appropriate. To start with, we consider the case when no RAG attributes are cached. The handling of cached attributes in combination with rewriting is treated in Section 6.3.

Rewrite rules for each node type are translated into a corresponding Java method, *rewriteTo()*, that checks rewrite rule conditions and returns the possibly rewritten tree. This method is iteratively invoked until no conditions are true. If all conditions in one node's rewriteTo() method are false, then rewriteTo() in the node's superclass is invoked. The generated Java method for the first example in Section 4 is shown below.

```
ASTNode Add.rewriteTo() {
  if(childType().equals(TypeSystem.STRING))
    return new StringAdd(leftOp(), rightOp())
  return super.rewriteTo();
}
```

To determine when no conditions are true and iteration should stop, a flag is set when the rewriteTo() method in ASTNode is reached, indicating that no overriding rewriteTo method has calculated a result tree. A flag is used since a simple comparison of the returned node is not sufficient because the rewrite may have rewritten descendent nodes only. In order to handle nested rewrites, a stack of flags is used.

Figure 5 shows an AspectJ aspect implementing the above described behaviour:

(1) The stack used to determine when no conditions are true
(2) Iteratively apply rewrite until no conditions are true
(3) Push *false* on the stack to guess that a rewrite will occur
(4) Bind the rewritten tree as a child to the parent node.
(5) Set top value on stack to *true* when rewriteTo in ASTNode is reached (no rewrite occurred)
(6) Define a pointcut when the child method is called.
(7) Each call to child is extended to also call rewrite.

```
   public aspect Rewrite {
(1)  protected static Stack noRewrite = new Stack();
(2)  ASTNode rewrite(ASTNode parent, ASTNode child,
                     int index) {
       do {
(3)        noRewrite.push(Boolean.FALSE);
           child = child.rewriteTo();
(4)        parent.setChild(index, child);
       } while(noRewrite.pop() == Boolean.FALSE);
       return child; }
(5)  ASTNode ASTNode.rewriteTo() {
       noRewrite.pop();
       noRewrite.push(Boolean.TRUE);
       return this; }
(6)  pointcut child(int index, ASTNode parent) :
       call(ASTNode ASTNode.child(int)) &&
       args(index) && target(parent);
(7)  ASTNode around (int index, ASTNode parent) :
           child(index, parent) {
       ASTNode child = proceed(index, parent):
       return rewrite(parent, child, index); }
   }
```

Figure 5. Aspect Rewrite: Iteratively rewrite each visited tree node

As discussed in Section 5.3 a tree node currently in rewrite may be visited again during that rewrite when reading attributes. When a node that is in rewrite is visited, the current tree state should be returned instead of initiating a new rewrite. That behaviour is implemented in the aspect shown in Figure 6:

(1) A flag, inRewrite, is added to each node to indicate whether the node is in rewrite or not.
(2) Add advice around each call to the rewriteTo method.
(3) The flag is set when a rewrite is initiated.
(4) The flag is reset when a rewrite is finished.
(5) Add advice around the rewrite loop in the previous aspect.
(6) When a node is in rewrite then the current tree is returned instead of initiating a new rewrite.

6.2 Optimization of final nodes

As mentioned, a node may be rewritten several times. We are interested in detecting when no further rewriting of it is possible so we know that it has

```
    public aspect ReVisit {
(1)   boolean ASTNode.inRewrite = false;
(2) ASTNode around(ASTNode child)
      : execution(ASTNode ASTNode+.rewriteTo())
      && target(child) {
(3)    child.inRewrite = true;
       ASTNode newChild = proceed(child);
(4)    child.inRewrite = false;
      return newChild; }
(5) ASTNode around(ASTNode child)
      : execution(ASTNode Rewrite.rewrite(ASTNode, ASTNode,
                                           int)
      && args(*, child, *) {
(6)    if(child.inRewrite)
        return child;
      return proceed(child);
    }
```

Figure 6. Aspect ReVisit: Pass through re-visit to a node already in rewrite

reached its final identity. By detecting final nodes, we can avoid the needless checking of their rewrite conditions (since they will all be false). This performance improvement can be significant for nodes with expensive conditions, e.g., when extracting a property by visiting all the children of the node. We can also use the concept of final nodes to cache attributes, as will be discussed in Section 6.3.

Definition 1. *A node is said to be* final *when i) all its rewrite conditions evaluate to false, and ii) future evaluations of its rewrite conditions cannot yield other values, and iii) it cannot be rewritten by any other node.*

Clearly, no further rewriting of final nodes is possible: i) and ii) guarantee that the node itself cannot trigger any rewriting of it, and iii) that it cannot be rewritten by any other node.

To find out when a node is final, we first recall (from Section 4) which nodes may be changed by a rewrite rule. Consider a node N which is the root of a subtree T. The rewrite rule will result in replacing T by T', where T' consists of a combination of newly created nodes and old nodes from T. I.e., the rewrite may not change nodes outside T. From this follows that a node can only be rewritten by rules in the node itself or rules in nodes on the path to the AST root node.

This allows us to state that

Lemma 1. *If a node is final, all its ancestor nodes are final.*

Proof. Otherwise the node may be rewritten by an ancestor node, in which case it is not final.

From Lemma 1 follows that at any point during evaluation, the final nodes of the AST will constitute a connected region that includes a path to the root, the *final region*. Initially, the evaluator visits only nodes in the final region, and is said to be in *normal* mode. But as soon as a non-final node is accessed from normal mode, the evaluator enters *rewrite* mode and that non-final node is said to be a *candidate*. When the iterative rewriting of the candidate has finished it turns out that it is final (see Theorem 2, and the evaluator returns to normal mode, completing the rewrite session. This way the final region is successively expanded. During a rewrite session, other non-final nodes may be visited and rewritten, but these are not considered candidates and will not become final during that rewrite session. There is only one candidate per rewrite session.

Note that during a rewrite session, the evaluator may well visit non-final nodes outside of the candidate subtree, and non-final nodes may be visited several times, the candidate included. For example, let us say we are rewriting a class String to add an explicit superclass reference to class Object. This means we will visit and trigger a rewrite of class Object. The rewrite of Object includes adding an explicit constructor. This involves searching through the methods of Object for a constructor. Suppose there is a method String toString() in Object. When this method is traversed, this will trigger rewriting of the identifier String to a type reference that directly refers to the String class. This in turn will involve a second visit to the String class (which was the candidate).

Theorem 2. *At the end of a rewrite session, the candidate c is final.*

Proof. At the end of the rewrite session, all rewrite conditions of c have just been evaluated to false. Furthermore, all ancestors of c are final, so no other node can rewrite c. What remains to be shown (see Definition 1) is that future evaluations of the rewrite conditions cannot yield other values. To see this we must consider the set of all other non-final nodes N that were visited in order to evaluate the rewrite conditions of c. This has involved evaluating all the rewrites conditions of these nodes in turn, also yielding false for all these conditions, and without triggering any rewrites of those nodes. Otherwise, another iteration of rewrite of c would have been triggered and we would not be at the end of the rewriting session. Since all these conditions evaluate to false, and there is no other node that can rewrite any of the nodes in N (since their ancestors outside N are final), none of these conditions can change value, and not only c, but in fact all nodes in N are final. □

In keeping track of which nodes are final, we add a flag isFinal to each node. In principle, we could mark both c and all the nodes in N as final at the end of the rewriting session. However, it is sufficient to mark c since any subsequent visits to a node in N will immediately mark that node as final, since all its rewrite conditions are false. An aspect introducing the isFinal flag is implemented in the aspect shown in Figure 7:

(1) A flag, isFinal, is added to each node to indicate whether the node is final or not.
(2) Add advice around the rewrite loop in the Rewrite aspect.
(3) When a node is final no rule condition checking is necessary and the node is returned immediately.
(4) When a node is entered during normal mode it becomes the next node to be final and we enter rewrite mode. On condition checking completion the node is final and we enter normal mode.
(5) A rewrite during rewrite mode continues as normal.

```
    public aspect FinalNodes {
(1)   boolean ASTNode.isFinal = false;
(2)   boolean normalMode = true;
(2)   ASTNode around(ASTNode parent, ASTNode child)
        : execution(ASTNode Rewrite.rewrite(ASTNode,
        ASTNode, int)) && args(parent, child, *) {
(3)     if(child.isFinal)
          return child;
(4)     if(normalMode) {
          normalMode = false;
          child = proceed(parent, child);
          child.isFinal = true;
          normalMode = true;
          return child; }
(5)     return proceed(parent, child); }
    }
```

Figure 7. Aspect FinalNodes: Detect final nodes and skip condition evaluation

6.3 Caching attributes in the context of rewrites

In plain RAGs, attribute caching can be used to increase performance by ensuring that each attribute is evaluated only once. When introducing rewrites the

same simple technique cannot be used. A rewrite that changes the tree structure may affect the value of an already cached attribute that must then be reevaluated. There are two principle approaches to ensure that these attributes have consistent values. One is to analyze attribute dependences dynamically in order to find out which attributes need to be reevaluated due to rewriting. Another approach is to cache only those attributes that cannot be affected by later rewrites. In order to avoid extensive run-time dependency analysis, we have chosen the second approach.

We say that an attribute is *safely cachable* when its value cannot be affected by later rewrites. Because final nodes cannot be further rewritten, an attribute will be safely cachable if all nodes visited during its evaluation are final.

A simple solution is to only cache attributes whose evaluation is started when the evaluator is in normal mode, i.e., not in a rewriting session. These attributes will be safely cachable. To see this, we can note that

i) the node where the evaluation starts is final (since the evaluator is in normal mode)

ii) any node visited during evaluation will be in its final form before its attributes are accessed, since any non-final node encountered will cause the evaluator to enter rewrite mode, returning the final node after completing that rewriting session.

It is possible to cache certain attributes during rewriting, by keeping track dynamically of if all visited nodes are final. However, this optimization has not yet been implemented.

As mentioned earlier, the ReRAG implementation is implemented as aspects on top of the plain RAG implementation. The RAG implementation caches attributes, so we need to disable the caching whenever not in normal mode in order to handle ReRAGs. This is done simply by advice on the call that sets the cached-flag. Figure 8 shows how this is done.

```
public aspect DisableCache {
  Object around() : set(boolean ASTNode+.*_cached) {
    if(!FinalNodes.normalMode)
      return false;
    return proceed(); }
}
```

Figure 8. Aspect DisableCache: Disable caching of attributes when not in normal mode

7 Implementation evaluation

7.1 Applicability

We have implemented ReRAGs in our tool JastAdd II and performed a number of case studies in order to evaluate their applicability.

Full Java static-semantics checker Our largest application is a complete static-semantic analyzer for Java 1.4. The grammar is a highly modular specification that follows the Java Language Specificaton [GJSB00], second edition, with modules like name binding, resolving ambiguous names, type binding, type checking, type conversions, inheritance, access control, arrays, exception handling, definite assignment and unreachable statements. An LALR(1) parser using a slightly modified grammar from the Java Language Specification [GJSB00], is used to build the initial abstract syntax tree. The AST is rewritten during the analysis to better capture the semantics of the program and simplify later computations. Some examples where rewrites were useful are:

- for resolving ambiguous names and for using semantic specialization for bound name references.
- for making implicit constructs explict by adding (as appropriate) empty constructors, supertype constructor accesses, type conversions and promotions, and inheritance from *java.lang.Object*.
- for eliminating shorthands such as splitting compound declarations of fields and variables to a list of single declarations.

Java to C compiler Our collegue, Anders Nilsson, has implemented a Java to C compiler in ReRAGs [Nil04], based on an older version of the Java checker. The generated C code is designed to run with a set of special C runtime systems that support real-time garbage collection, and is interfaced to through a set of C macros. ReRAGs are used in the back end for adapting the AST to simplify the generation of code suitable for these runtime systems. For example, all operations on references are broken down to steps of only one indirection, generating the macro calls to the runtime system. ReRAGs are also used for optimizing the generated code size by eliminating unused classes, methods, and variables. They are also used for eliminating shorthands, for example to deal with all the variants of loops in Java.

Worst-case execution time analyzer The Java checker was extended to also compute worst-case execution times using an annotation mechanism. The extension could be done in a purely modular fashion.

Automation Language The automation language *Structured Text* in the IEC-61131-3 standard has been modeled in ReRAGs and extended with an

object-oriented type system and instance references. The extended language is translated to the base language by flattening the class hierarchies using iterative rewriting. Details will appear in a forthcoming paper.

7.2 Performance

We have implemented ReRAGs in our aspect-oriented compiler compiler tool JastAdd II. To give some initial performance measurements we benchmark our largest application, a complete static-semantic analyzer for Java 1.4. After parsing and static-semantic analysis the checked tree is pretty printed to file. Since code generation targeted for the Java virtual machine, [LY99], is fairly straight forward once static-semantic analysis is performed we believe that the work done by our analyzer is comparable to the work done by a java to byte-code compiler. We therefore compare the execution time of our analyzer to the standard java compiler, javac, in J2SE JDK.

Two types of optimizations to the basic evaluation algorithm were discussed in Section 6.2 and Section 6.3. The first disables condition checking for nodes that are final and the second caches attribute values that only depend on attributes in final nodes. To verify that these optimizations improve performance we benchmark our analyzer with and without optimizations. The execution times when analysing a few files of the *java.lang* package are shown in Figure 9. These measurements show that both attribute caching and condition checking disabling provide drastic performance improvements when applied individually and even better when combined. Clearly, both optimizations should be used to get reasonable execution times.

The execution times do not include parsing that took 3262ms without attribute caching and slightly more, 3644ms, when caching attributes. We believe the increase is due to the larger tree nodes used when caching attributes.

	condition checking	no condition checking
no attribute caching	546323 ms	61882 ms
attribute caching	21216 ms	2016 ms

Figure 9. Comparison of analysis execution time with and without optimizations

To verify that the ReRAG implementation scales reasonably we compare execution times with a traditional Java compiler, javac, see Figure 10. We are using a subset of the Java class library, the *java.lang, java.util, java.io* packages, as our benchmarks. Roughly 100.000 lines of java source code from J2SE

JDK 1.4.2 are compiled, and the ReRAG-based compiler uses both the optimizations mentioned above. The comparison is not completely fair because javac generates byte code whereas the ReRAG compiler only performs static-semantic analysis and then pretty-prints the program. However, generating byte code from an analyzed AST is very straight-forward and should be roughly comparable to pretty-printing. The comparison shows that the ReRAG-based compiler is only a few times slower than javac. Considering that the ReRAG-based compiler is generated from a declarative specification, we find this highly encouraging. This shows that ReRAGs are already useful for large-scale practical applications.

	total	JVM init	parsing	analysis and prettyprinting
ReRAG compiler	22801ms	600ms	7251ms	14950ms
javac	6112ms			

Figure 10. Compile time for the *java.lang, java.util, java.io* packages using the ReRAG-based compiler and javac.

8 Related work

Higher-ordered Attribute Grammars ReRAGs are closely related to Higher-ordered Attribute Grammars (HAGs) [VSK89], [Sar99] where an attribute can be *higher-order*, in that it has the structure of an AST and can itself have attributes. Such an attribute is also called an *ATtributable Attribute* (ATA). Typically, there will be one equation defining the bare AST (without attributes) of the ATA, and other equations that define or use attributes of the ATA, and which depend on the evaluation of the ATA equation.

In ReRAGs each node in the AST is considered to be the root of a *rewritable attribute* of its parent node and may be rewritten to an alternative subtree during attribute evaluation. The rewriting is done conditionally, in place (replacing the original subtree during evaluation), and may be done in several steps, each described by an individual rewrite rule. This is contrast to the ATAs of HAGs which are constructed unconditionally, in one step, and where the evaluation does not change previously existing parts of the AST (the new tree is stored as a previously unevaluated attribute).

A major difference lies in the object-oriented basis of ReRAGs, where reference attributes are kept as explicit links in the tree and subtrees are

rewritten in place. HAGs, in contrast, have a functional programming basis, viewing the AST as well as its attributes as structured values without identity. This is in our view less intuitive where, for instance, cross references in the AST have to be viewed as infinite values.

HAGs + Forwarding Forwarding [VWMBK02] is an attribute grammar technique used to forward attribute equations in one node to an equation in another node. This is transparent to other attribute equations and when combined with ATAs that use contextual information it allows later computations to be expressed on a more suitable model in a way similar to ReRAGs. To simulate a nested and multi-level rewrite there would, however, conceptually have to be a new tree for each step in the rewrite.

Visitors The Visitor pattern is often used in compiler construction for separation of concerns when using object-oriented languages. Visitors can only separate cross-cutting methods while the weaving technique used in JastAdd can be used for fields as well. This is superior to the Visitor pattern in that there is no need to rely on a generic delegation mechanism resulting in a cleaner more intuative implementation and also provide type-safe parameter passing during tree traversal. ReRAGs also differ in that traversal strategies need not be specified explicitly since they are implicitly defined by attribute dependences. The use of attributes provide better separation of concerns in that contextual information need not be included in the traversal pattern but can be declared separately.

Rewrite Systems ReRAGs also have similarities to tree transformation systems like *Stratego* [Vis01b], *ASF+SDF* [vdBea01], and *TXL* [Cor04] but improves data acquisition support through the use of RAGs instead of embedding contextual data in rewrite rules or as global variables. *Stratego* uses Dynamic Rewrite Rules [Vis01a] to separate contextual data acquisition from rewrite rules. A rule can be generated at run-time and include data from the context where it originates. That way contextual data is included in the rewrite rule and need not be propagated explicitly by rules in the grammar. ReRAGs provide an even cleaner separation of rewrite rule and contextual information by the use of RAGs that also are superior in modeling complex non-local dependences. The rewrite application order differs in that ReRAGs only support the described declarative approach while the other systems support user defined strategies. In *Stratego* and *ASF+SDF* the user can define explicit traversal strategies that control rewrite application order. Transformation rules in *TXL* are specified through a pattern to be matched and a replacement to substitute for it. The pattern to be matched may be guarded by conditional rules and the replacement may be a defined as a function of the matched pattern. A function used in a transformation rule may in turn be a composed from other

functions. The rewrite application strategy in *TXL* is thus implicitly defined as part of the functional decomposition of the transformation ruleset, which controls how and in which order subrules are applied. Dora [BFG92] supports attributes and rewrite rules that are defined using pattern matching to select tree nodes for attribute definitions, equation, and as rewrite targets. Attribute equations and rewrite results are defined through Lisp expressions. Composition rules are used to define how to combine and repeat rewrites and the order the tree is traversed. The approach is similar to ReRAGs in that attribute dependences are computed dynamically at runtime but there is no support for remote attributes and it is not clear how attributes read during rewriting are handled.

Dynamic reclassification of objects Semantic specialization is similar to dynamic reclassification of objects, e.g. Wide Classes, Predicate Classes, FickleII, and Gilgul. All of these approaches except Gilgul differ from ReRAGs in that they may only specialize a single object compared to our rewritten sub-trees. *Wide Classes* [Ser99] demonstrates the use of dynamic reclassification of objects to create a more suitable model for compiler computations. The run-time type of an object can be changed into a super- or a sub-type by explicitly passing a message to that object. That way, instance variables can be dynamically added to objects when needed by a specific compiler stage, e.g., code optimization. Their approach differs from ours in that it requires run-time system support and the reclassification is explicitly invoked and not statically type-safe. In *Predicate Classes* [Cha93], an object is reclassified when a predicate is true, similar to our rewrite conditions. The reclassification is dynamic and lazy and thus similar to our demand-driven rewriting. The approach is, however, not statically type-safe. *FickleII* [DDDCG02] has strong typing and puts restrictions on when an object may be reclassified to a super type by using specific state classes that may not be types of fields. This is similar to our restriction on rewriting nodes to supertypes as long as they are not used in the right hand side of a production rule as discussed in Section 4.2. The reclassification is, however, explicitly invoked compared to our declarative style. *Gilgul* [Cos01] is an extension to Java that allows dynamic object replacement. A new type of classes, implementation-only classes, that can not be used as types are introduced. Implementation-only instance may not only be replaced by subclass instances but also by instances of any class that has the same least non implementation-only superclass. Object replacement in Gilgul is similar to our appraoch in that no support from the run-time system is needed. Gilgul uses an indirection scheme to be able to simultaneously update all object references through a single pointer re-assignment. The ReRAGs implementation uses a different approach and ensures that

all references to the replaced object structure are recalculated dynamically on demand.

9 Conclusions and Future Work

We have introduced a technique for declarative rewriting of attributed ASTs, supporting conditional and context-dependent rewrites during attribution. The generation of a full Java static-semantic analyzer demonstrates the practical use of this technique. The grammar is highly modular, utilizing all three dimensions of separation of concerns: inheritance for separating the description of general from specific behavior of the language constructs (e.g., general declarations from specialized declarations like fields and methods); aspects for separating different computations from each other (e.g., type checking from name analysis); and rewriting for allowing the computations to be expressed on suitable forms of the tree. This results in a specification that is easy to understand and to extend. The technique has been implemented in a general system that generates compilers from a declarative specification. Attribute evaluation and tree transformation are performed automatically according to the specification.The running times are sufficiently low for practical use. For example, parsing, analyzing, and prettyprinting roughly 100.000 lines of Java code took approximately 23 seconds as compared to 6 seconds for the javac compiler on the same platform.

We have identified several typical ways of transforming an AST that are useful in practice: Semantic Specialization, Make Implicit Behavior Explicit, and Eliminate Shorthands. The use of these transformations has substantially simplified our Java implementation as compared to having to program this by hand, or having to use a plain RAG on the initial AST constructed by the parser.

Our work is related to many other transformational approaches, but differs in important ways, most notably by being declarative, yet based on an object-oriented AST model with explicit references beween different parts. This gives, in our opinion, a very natural and direct way to think about the program representation and to describe computations.

Many other transformational systems apply transformations in a predefined sequence, making the application of transformations imperative. In contrast, the ReRAG transformations are applied based on conditions that may read the current tree, resulting in a declarative specification.

There are many interesting ways to continue this research.

Optimization The caching strategies currently used can probably be improved in a variety of ways, allowing more attributes to be cached, resulting in better performance.

Termination Our current implementation does not deal with possible non-termination of rewriting rules (i.e., the possibility that the conditions never become false). In our experience, it can easily be seen (by a human) that the rules will terminate, so this is usually not a problem in practice. However, techniques for detecting possible non-termination, either statically from the grammar or dynamically, during evaluation, could be useful for debugging.

Circular ReRAGs We plan to combine earlier work on CRAGs [MH03] with our work on ReRAGs. We hope this can be used for running various fixed-point computations on ReRAGs, with applications in static analysis.

Language extensions Our current studies on generics indicate that the basic problems in GJ [BOSW98] can be solved using ReRAGs. Extending our Java 1.4 to handle new features in Java 1.5 like generics, autoboxing, static imports, and type safe enums is a natural next step.

Acknowledgements

We are grateful to John Boyland and to the other reviewers (anonymous) for their valuable feedback on the first draft of this paper.

References

[BFG92] John Boyland, Charles Farnum, and Susan L. Graham. Attributed transformational code generation for dynamic compilers. In R. Giegerich and S. L. Graham, editors, *Code Generation - Concepts, Tools, Techniques. Workshops in Computer Science*, pages 227–254. Springer-Verlag, 1992.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of Object Oriented Programming: Systems, Languages, and Applications (OOPSLA*, pages 183–200, 1998.

[Cha93] Craig Chambers. Predicate classes. In *Proceedings of ECOOP'93*, volume 707 of *LNCS*, pages 268–296. Springer-Verlag, 1993.

[Cor04] James R. Cordy. Txl: A language for programming language tools and applications. In *Proceedings of the 4th Workshop on Language Descriptions, Tools, and Applications (LDTA'04) at ETAPS 2004*, 2004.

[Cos01] Pascal Costanza. Dynamic object replacement and implementation-only classes. In *6th International Workshop on Component-Oriented Programming (WCOP 2001) at ECOOP 2001*, 2001.

[DDDCG02] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. More dynamic object reclassification: FickleII;. *ACM Trans. Program. Lang. Syst.*, 24(2):153–191, 2002.

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

[Hed89] Görel Hedin. An object-oriented notation for attribute grammars. In *the 3rd European Conference on Object-Oriented Programming (ECOOP'89)*, pages 329–345. Cambridge University Press, July 1989.

[Hed00] Görel Hedin. Reference Attributed Grammars. *Informatica (Slovenia)*, 24(3), 2000.

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

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *LNCS*, 2072:327–355, 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).

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[MH03] Eva Magnusson and Görel Hedin. Circular reference attributed grammars - their evaluation and applications. *Electronic Notes in Theoretical Computer Science*, 82(3), 2003.

[Nil04] Anders Nilsson. Compiling Java for Real-Time Systems. Licentiate thesis, Department of Computer Science, Lund Institute of Technology, 2004.

[Sar99] Joao Saraiva. *Purely functional implementation of attribute grammars*. PhD thesis, Utrecht University, The Netherlands, 1999.

[Ser99] Manuel Serrano. Wide classes. In *Proceedings of ECOOP'99*, volume 1628 of *LNCS*, pages 391–415. Springer-Verlag, 1999.

[vdBea01] M. van den Brand et al. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of Compiler Construction Conference 2001*, volume 2027 of *LNCS*. Springer-Verlag, 2001.

[Vis01a] Eelco Visser. Scoped dynamic rewrite rules. *Electronic Notes in Theoretical Computer Science*, 59(4), 2001.

[Vis01b] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Proceedings of Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, 2001.

[VSK89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the SIGPLAN '89 Programming language design and implementation*. ACM Press, 1989.

[VWMBK02] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proceedings of Compiler Construction Conference 2002*, volume 2304 of *LNCS*, pages 128–142. Springer-Verlag, 2002.

Paper II

The JastAdd System - modular extensible compiler construction

Torbjörn Ekman and Görel Hedin

Department of Computer Science, Lund University, Sweden
(torbjorn|gorel)@cs.lth.se

The JastAdd System – modular extensible compiler construction

Torbjörn Ekman and Görel Hedin

Department of Computer Science, Lund University, Sweden
(torbjorn|gorel)@cs.lth.se

Abstract The JastAdd system enables open modular specifications of extensible compiler tools and languages. Java has been extended with the Rewritable Circular Reference Attributed Grammars formalism that supports modularization and extensibility through several synergistic mechanisms. Object-orientation and static aspect-oriented programming are combined with declarative attributes and context-dependent rewrites to allow highly modular specifications. The techniques have been verified by implementing a full Java compiler.

1 Introduction

We present the JastAdd system for development of compilers and related tools. The system enables open modular specification of extensible compiler tools and languages. JastAdd is an extension to Java that supports a specification formalism called Rewritable Circular Reference Attributed Grammars (ReCRAGs). ReCRAGs raises the abstraction level for grammar-based computations using declarative object-oriented techniques to rewrite abstract syntax trees.

Several synergistic mechanisms are combined to better support modularity and extensibility; object-orientation, static aspects, declarative attributes, and context-dependent rewrites. Object-orientation gives modularization benefits like class hierarchies where behavior can be specialized and overridden. Static aspect-oriented programming in the form of intertype declarations is used to decouple behavior from the base class descriptions. Declarative attributes are used when specifying behavior, allowing a high degree of decoupling between different language constructs. Context-dependent rewrites allow the model, in this case that abstract syntax tree (AST), to be dynamically changed to better suit different computations, allowing improved modularization of many computations.

The order in which the attribute computations and rewrites are carried out is decided automatically through their implicit dependences instead of requiring

explicit ordering in the code. This implicit scheduling of computation is a major reason why such a high degree of decoupling of the modules is possible.

The combination of these techniques enables specifications to be written in a highly modular fashion which is important both for understanding the specification, by breaking it down into simple steps, and for reuse. These mechanisms also enable decomposition of the system according to different criteria, such as functionality or language elements. For example, the static-semantic analysis of a compiler can be reused in a source code metrics tool or extended to support a new version of the language.

We have verified that the techniques scale to real languages by implementing a full Java compiler using JastAdd. We have validated the generated compiler against the Jacks test suite[1], and it passes more tests than both the popular javac and jikes compilers. The size of the specification is roughly half the size of the handwritten javac implementation and the generated compiler's performance is within a factor of four. The integration of the declarative attributes and rewriting techniques with Java and its commonly used object-oriented features, makes the system fairly easy to learn for software developers in industry, thus not restricting its use to the computer science research community. Indeed, the system has been used sucessfully in several undergraduate projects as well as in industry.

The rest of this paper is structured as follows. Section 2 introduces the intended application domain through a case study of a full java compiler implemented in JastAdd. Section 3 describes how various features can be used to specify modular and extensible descriptions. Section 4 briefly explains the evaluation method used. Section 5 compares our system to related tools and systems. Section 6 concludes the paper.

2 Application domain

The primary application domain for the JastAdd system is extensible compilers and analysis tools. It has been used to implement full languages like Java; little languages such as toy languages in an undergraduate course; as well as domain specific languages such as robotics and automation languages. We have verified that the techniques scale to real languages by implementing a full Java compiler using JastAdd and comparing it to popular handwritten compilers, as mentioned above. The Java compiler has been modularized according to several different criteria as a proof of concept, but also to build unrelated tools that share common infrastructure.

The basis for the compiler is a front-end that performs static-semantic analysis for Java 1.4. This front-end is modularized according to the various concerns in the analysis, e.g., name resolution, type checking, definite assignment,

unreachable statements, and exception handling. It can be used as a stand-alone tool and has also been extended with various experimental analysis modules such as devirtualization and metrics. The front-end is used by multiple back-ends that target different run-time environments [2]: byte code for Java Virtual Machines, byte code for the Palcom Runtime Environment[3], and C source code. The initial decomposition criterion was functionality, but we have also extended the compiler with new language constructs that crosscut existing modules. The Java1.4 compiler, consisting of the front-end and java bytecode back-end, has been extended with complex language features from Java 1.5, including generics with wildcards that extend the type system significantly, as well as with features from AspectJ, including intertype declarations that include highly non-local changes to name resolution. All extensions were implemented in a completely modular, yet understandable way.

3 Features and Foundation

The JastAdd tool takes a set of components as input and generates ordinary Java code. A component consists of abstract grammar modules, declarative ReCRAG modules, and imperative Java modules. From these specification modules the tool generates AST classes that contain woven generated code. The AST is built using any Java-based parser generator, e.g., JavaCC, ANTLR, CUP, or Beaver, using node constructors in the semantic actions.

There are numerous features in JastAdd that enable modular specifications. This section describes how they can be combined to further enhance modularity and extensibility. We also exemplify how they are used in our Java compiler implementation. Please consult the system distribution for documentation of the actual tool syntax and tutorial examples. In addition to the Java compiler, the distribution contains a number of smaller examples that illustrate and document typical use of the features.

3.1 Basic design

Attribute Grammars (AG) [4] have proven useful when describing context sensitive information for programming languages. Their declarativeness makes it easy to modularize grammars freely. They also integrate well with the object-oriented programming paradigm both from a description and evaluation point of view as shown in [5,6]. The main design behind JastAdd is to integrate object-oriented AGs and Java and use the AST as the only data structure. Several extensions to AGs have been combined into the system, to scale this design to real languages. The AST is modelled as an object-oriented class hierarchy

that allows behavior to be specialized. In combination with intertype declarations [7] (also known as open classes [8]), this enables good separation of concerns between different computations. The computations can be further decoupled by expressing the computations using declarative attributes thus not requiring explicit ordering of computations.

Synthesized attributes are, in the object-oriented notation for AGs, very similar to traditional virtual methods in object-oriented programming. The main difference is that synthesized attributes must be functions that may not have side-effects. This allows for efficient evaluation through caching of the result. That same result is then returned for each occurrence of that attribute in other equations. The use of synthesized attributes decouples the specification of an attribute from its actual implementation.

The use of inherited attributes decouples an AST node from its parent: the AST node does not need to know which parent it has. All the information it needs is in the inherited attributes whose values are defined by the parent. This allows AST classes with all their behavior to be reused in many different contexts. For example, expressions may occur inside many different kinds of statements and declarations. The behavior of the expression depends only on its inherited attributes, not on any specific surrounding node.

3.2 References add additional structure to the AST

Reference Attributed Grammars (RAGs) [9,10,11] allow graphs to be represented on top of the AST and allow attribute dependences to follow such graphs rather than the tree structure. References can then be used to model various language relations as bindings. Name resolution becomes the task of binding a name to its declaration. Types can, in a similar way, be represented by their declarations and then the type of an expression is simply a reference from an expression node to a type declaration node. This reference can then be used to compute further properties, e.g., binding names to members in structured types.

References are used extensively in the Java compiler to model various graph structures such as the inheritance hierarchy, recursive types, and the call graphs. Relations can also be modelled as sets of references. For example, a method overrides/hides a set of other methods; a type is a subtype of a set of types. References are not only used for names and types but also for modelling non-local transfer of control such as break, continue, and exception handling. Break and continue are bound to a possibly labeled destination statement while thrown exceptions are bound to a catch clause or a method that throws a compatile exception. Each control-changing statement also holds a list of references to possible finally clauses that are visited on its way to its destination node.

3.3 Broadcasting scope and parameterized attributes

Inherited attributes decouple the AST node from its parent. This can be improved further by decoupling the parent from its child. Inherited attributes can be broadcasted not only to a single child but to all its descendants. This feature is very similar to the *including* feature in [12]. The scope for the attribute equation is then widened from a single node to an entire subtree. The only requirement for an inherited attribute is then that there is a path from the declaration of an attribute through its ancestors to an equation. By allowing a descendant to redefine the equation for its subtree, the broadcasting mechanism is an excellent way to describe nested scopes. It is highly desirable from a modularization point of view that the modules defining scoped data need not be aware of the modules using the scoped data and vice versa. This makes it easy to extend the language with new constructs, both constructs that introduce new scopes, and constructs that make use of scoped data.

This loose coupling can further be enhanced by allowing attributes to have parameters. For example, the set of visible variables can be parameterized by a name. Information can then flow both upwards, through the argument, and downwards, by the broadcasted attribute, while keeping the modularization benefits described above.

Broadcasting is used in the Java compiler to compute name binding and many of the other relations described in the previous section. The attribute is often parameterized with a name. This is used to broadcast scope to bind various names in a modular way, e.g., variables, methods, types, and labels. Broadcasting is also used to describe nesting in a simple way. For example, enclosing type declaration, enclosing body declaration, and enclosing compilation unit.

3.4 Complex non-local dependences

The superimposed graph structures on top of the AST, in combination with declarative attributes, makes it easy to decompose a complex computation into several simpler ones. A combination of broadcasting of parameterized attributes and delegation through a reference allows complex, highly non-local, dependences to be expressed is a simple fashion. Object-oriented inheritance can, for instance, be implemented by using broadcasting for nested scopes combined with delegation to the superclass through a reference. If a member declaration is not found inside a class, the class declaration node can delegate the search to its superclasses. This implementation matches traditional object-oriented inheritance with overriding. The scheme can be extended to support nested types through delegation to the enclosing broadcast when the superclass search is fruitless. This supports, for instance, modelling of inner classes.

This technique has been used to implement the entire name resolution and binding part in the Java compiler. The combination described above is combined with filters to handle access control. Qualified names, such as *collection.size()*, are implemented by delegating the binding for the right hand side, *size()*, to the type of the left hand side, *collection*, which may be *java.util.-Collection*. The scheme has also been extended in a modular fashion to support intertype declarations as in AspectJ or Open Classes.

3.5 Enhance tree structure

The initial context-free AST, often built by a parser, is not a perfect match for most computations. Since the AST is the only data structure in the system it would be quite limiting if the AST could not be changed. Rewritable RAGs (ReRAGs) allow the initial AST to be rewritten to a more suitable form, based on values of the context-dependent attributes. This allows the tree to not only reflect context-free structure but also the context-sensitive information that has been computed so far. For example, a name node can be rewritten to a specialized node to reflect its semantic meaning, like a field- or type-name, as soon as that information is available. This allows later computations, like optimization and code generation, to be modularized according to the semantic meaning of the name. For example, splitting the generation of code for field accesses and type-name accesses into separate modules. Rewrites can thus be used to make the tree more suitable for the other modularization features in JastAdd. The rewriting is done iteratively, interleaved with attribute evaluation. This allows complex rewrites to be broken down into many simple rewrites. The fine-grained interaction between rewrites and attribute evaluation is further discussed in [13].

Rewrites can also be used to normalize the AST into a kernel language. This can be used for rewriting language extensions to constructs in the base language, something which often requires contextual information. Yet another use of context-dependent rewrites is to separate language constructs that are ambiguous from a context-free perspective, such as the infamous typedef vs. variable ambiguity in C.

The Java compiler uses rewrites to make implicit language behavior explicit in the AST. For example, to add default constructors to classes with no constructors, and to add an explicit *this* to unqualified invocations of instance methods. A series of rewrites are also used to resolve syntactically ambiguous names. Names are initially reclassified according to their position in the grammar. For example, the name after an extends clause is expected to be a type name. However, if the name is qualified, e.g., *a.b*, we only know that *b* is a type name and *a* can either be a package name or a type name. That name is

thus ambiguous and can be further reclassified by taking visible declarations into account. The name is reclassified to a type name if there is a visible type declaration named *a*, and otherwise to a package name. Multiple names are then grouped to form a semantic unit. For example, the individual parts of a package name and a type name are combined into a fully qualified type name. The use of rewrites for name analysis is described in more detail in [14]. Other uses of rewrites include splitting variable declarations with multiple variables and grouping scattered cardinality for array names.

3.6 Non terminal attributes

It is sometimes useful to expand an AST with additional nodes that are defined by equations, rather than constructed by the parser or by a rewrite rule. These nodes are called nonterminal attributes (NTAs) [15,16] since they are both similar to nodes (nonterminals) and to attributes. An NTA is like a node in that it can itself have attributes and it can be rewritten. It is also like an attribute in that it is defined by an equation. Grammars with NTAs are considered higher-order attribute grammars since attributes may themselves have attributes. There are similarities between NTAs and context-dependent rewrites. In both cases, you can declaratively define changes to the AST based on attribute values. The main difference is that you should use rewrites when you are interested in replacing some nodes with others, and NTAs when you want to keep existing nodes and introduce some additional ones. NTAs in combination with a technique called attribute forwarding [17] is similar to replacing a tree by a new one.

NTAs are used in the Java compiler to add predefined declarations, such as primitive types, but more importantly to instantiate generic types in Java 1.5. The NTA is then a function of the generic type declaration parameterized with one or more type parameters.

3.7 Circular attributes

JastAdd supports circular attributes [18] which are useful for many analysis problems, for example in code optimization. Attributes may then be mutually dependent and the evaluator computes fixed-point solutions by iteration. The use of circular attributes is particularly useful in combination with reference attributes, since they can then be used to compute mutually recursive properties on top of graphs [19].

We have used circular attributes in the Java compiler to detect circularities in inheritance hierarchies and to determine definite assignment properties of variables in loop constructs.

3.8 Imperative modules

Aspect modules with static introductions can be used not only for declarative attributes but also for imperative Java code. Methods and fields can then be introduced in existing classes in a modular fashion. Imperative code may use declarative attributes by invoking them as if they were methods. Attribute evaluation and rewriting of the AST is transparent to the imperative code. The back-end of the Java compiler is a combination of declarative attributes and an imperative traversal that outputs the tree to file.

4 The JastAdd evaluation engine

The evaluation algorithm in JastAdd is built on lazy dynamic evaluation with attribute caching. Dependences are computed dynamically and on demand. Since references may reference arbitrary nodes it is difficult (probably impossible in general) to schedule the order statically. Lazy evaluation has some nice properties for the execution time of attribute computations; only attributes that are being used add cost to the execution time. For example, attributes used to compute error messages do not add to the overall execution time as long as there are no errors in the analyzed program. Similarly, an attribute can be defined for a large set of node types even though only a few nodes actually depend on the attribute value. This is useful for obtaining a simple specification, and yet does not incur additional cost. For example, references to primitive types can be broadcast throughout an entire AST, but may actually be used in only a few places.

Conditional rewriting is also done lazily and is triggered implicitly when a node is visited. When evaluating the condition for a rewrite, other nodes may be visited. This will in turn trigger further rewrites. When no conditions are true for rewrites in the visited nodes, the resulting tree node is returned. A nice property of this evaluation technique is that a traversal of the tree need not be aware of rewriting since the returned node is always a node for which no rewrite conditions are true. A thorough description of the evaluation of rewrites and its interaction with attribute evaluation is given in [13].

The combination of lazy evaluation and attribute caching works very well in practice. The execution speed of our generated Java compiler is well within a factor of four compared to the handwritten javac compiler.

5 Related tools and systems

The current version of the JastAdd system builds loosely on an older version [20] which supported RAGs and intertype declarations. The current JastAdd

system has added support for rewrites, parameterized attributes, broadcasting, nonterminal attributes, and circular attributes, thereby enabling the concise implementation of real programming languages like Java. The current system is also bootstrapped in itself.

5.1 Imperative systems

The primary application domain for the JastAdd system is extensible compilers and analysis tools. Our largest specification, the Java compiler, can be compared to handwritten extensible Java compilers. The Polyglot system [21,22], is a Java 1.4 front end supporting extending Java with new language constructs, translating them to Java source code. In contrast to JastAdd, the extensions in Polyglot are coded imperatively, making use of variants of the visitor design pattern. A phase-oriented architecture with fixed AST traversals is used, so that different computations need to be explicitly associated with different phases, and it is the burden of the user to make sure that everything is computed in the appropriate order. Preliminary experiments with extending Java with AspectJ-like constructs indicate that using JastAdd leads to much more concise and clear specifications as well as to a faster translation tool.

JastAdd is used to specify contextual computations on top of an AST. The initial AST is usually built using a parser generator, e.g., JavaCC, ANTLR, CUP, SableCC, or Beaver, using node constructors in semantic actions. Some of these tools support limited contextual computations through the use of visitors or semantic actions. Their imperative nature make these computations inferior to JastAdd modules from a modularity and extensibility point of view.

5.2 Attribute grammar systems

There are many other attribute grammar systems, both commercial and licensed systems like the Synthesizer Generator [23] and Cocktail [24], as well as freely available systems like Eli [25], Elegant [26], LRC [27], and UAG [28]. While all of these systems support synthesized and inherited attributes, and many of them nonterminal attributes through higher-ordered grammars, there are many differences as compared to JastAdd.

One important difference is that most of them do not support reference attributes. One exception is the Elegant system that supports a notion similar to reference attributes which is used for name bindings, but via a global symbol-table data structure. Cocktail has a concept called tree-valued attributes which also seems similar to reference attributes, but we have not found any examples that show how they are used. In systems based on non-strict functional languages, like UAG, it should in principle be possible to use lazy evaluation

to emulate reference attributes. However, we have not seen any documented examples that take advantage of this. In JastAdd, reference attributes constitute the key mechanism to deal with non-local dependences, not only for name bindings. AG systems that do not use reference attributes need to encode the context into attributes and pass them around explicitly, resulting in coupled specifications.

A very important difference between JastAdd and other systems is the support of context-dependent rewrites interleaved with attribute computations. This is a key mechanism that allows complex analysis problems like Java name resolution and type analysis to be broken down into small simple steps. To our knowledge, there are no other systems supporting similar mechanisms. Nonterminal attributes combined with forwarding [17] would be similar, but as far as we know, forwarding has only been implemented in prototypes built on top of Haskell, and it is unclear how the practical performance would scale to full languages like Java.

Another difference between JastAdd and the other mentioned AG systems is the support for circular attributes and parameterized attributes, which is not implemented in any of the other systems.

From a software engineering perspective there is an important difference between JastAdd and other AG systems in the integrated use of Java. The syntax for JastAdd specifications is a superset of Java, and constructs for attributes and equations are very Java-like, allowing users to think of attributes and equations as method declarations and method implementations. The aspect-oriented syntax used in JastAdd is very similar to that used in AspectJ. This is different from most other AG systems which use their own specific syntax for the attribution. This integration makes it straight-forward to combine the declarative computations with imperative mainstream object-oriented programming in Java, and makes it easy for Java programmers to learn the tool and the concepts.

5.3 Transformation Systems

The main focus of JastAdd is context-dependent computations on the AST, but since rewriting is also supported we compare also to tree-transformation systems that are used for generating language-based tools, e.g., ASF+SDF [29], Stratego [30], and TXL [31].

An important difference between these systems and JastAdd is the way to specify the order of transformations. Transformation systems specify the order using implicit predefined traversals or explicit user-defined strategies. JastAdd, on the other hand, uses fine-grained attribute dependences to drive the traversal which in turn implicitly defines the order of transformations. Complex traversal patterns can thus automatically follow dependences and do not have to be stated explicitly.

Another important difference is that transformation systems typically handle contextual information by using an external database that is updated during the transformations. This requires the user to explicitly associate database updates with particular transformation rules or phases. The traversal order must thus take contextual dependences, which can be highly non local, into account. In contrast, JastAdd uses the contextual dependences to derive a suitable traversal strategy. The Stratego system has a mechanism for dependent dynamic transformation rules [32], supporting certain context-dependent transformations, but it is not clear how this could be used for implementing name binding and similar problems in object-oriented languages.

6 Conclusions

We have presented the JastAdd tool and shown how it supports implementation of extensible compiler tools and languages. A key design idea is to make use of declarative specification mechanisms in order to allow a high degree of decoupling between different modules, thereby supporting reuse and extensibility. Another key design idea is to build on object-orientation and Java, thereby both taking advantage of the support for modelling and reuse available in object-orientation, as well as making the declarative techniques easily understood by Java programmers.

In addition to well-known specification features like inherited, synthesized, and nonterminal attributes, JastAdd includes the very powerful feature of context-dependent rewrites, allowing the AST to be modified taking context-sensitive computations into account. A key feature is also that of reference attributes, allowing the AST itself to be used as the fact database. Additional JastAdd features like parameterized attributes and circular attributes also contribute substantially to the decoupling of modules and computations. Small examples are provided that demonstrate the typical use of all the features.

To demonstrate the full power of the tool we have successfully implemented a very strong case: a complete Java 1.4 compiler including compile-time checks and bytecode generation. Java 1.4 is a large complex language and implementing a complete compiler for it is a substantial undertaking, both because the language contains many idiosyncrasies that must be handled, and because it is an object-oriented language with many non-trivial constructs. We are not aware of any other declarative implementation of a complete practical object-oriented language. To demonstrate extensibility both for language constructs and tool functionality, we have extended the Java 1.4 compiler with new language constructs from Java 1.5, and extended the Java 1.4 front end with devirtualization analysis. All these extensions have been done in a completely modular way.

References

1. The Jacks compiler test suite, http://sources.redhat.com/mauve/ (2006).
2. A. Nilsson, A. Ive, T. Ekman, G. Hedin, Implementing Java Compilers using ReRAGs, Nordic Journal of Computing 11 (3) (2004) 213–234.
3. Palpable Computing, http://www.ist-palcom.org (2006).
4. D. E. Knuth, Semantics of context-free languages, Mathematical Systems Theory 2 (2) (1968) 127–145, correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).
5. G. Hedin, An object-oriented notation for attribute grammars, in: the 3rd European Conference on Object-Oriented Programming (ECOOP'89), BCS Workshop Series, Cambridge University Press, 1989, pp. 329–345.
6. K. Koskimies, Object-orientation in attribute grammars, in: Proceedings on Attribute Grammars, Applications and Systems, Springer-Verlag, London, UK, 1991, pp. 297–329.
7. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, Lecture Notes in Computer Science 2072 (2001) 327–355.
8. C. Clifton, G. T. Leavens, C. Chambers, T. Millstein, MultiJava: Modular open classes and symmetric multiple dispatch for Java, in: Proceedings of OOPSLA 2000, Vol. 35(10), 2000, pp. 130–145.
9. G. Hedin, Reference Attributed Grammars, in: Informatica (Slovenia), 24(3), 2000, pp. 301–317.
10. A. Poetzsch-Heffter, Prototyping realistic programming languages based on formal specifications, Acta Informatica 34 (1997) 737–772.
11. J. T. Boyland, Descriptional composition of compiler components, Ph.D. thesis, University of California, Berkeley, available as technical report UCB//CSD-96-916 (Sep. 1996).
12. U. Kastens, W. M. Waite, Modularity and reusability in attribute grammars, Acta Informatica 31 (7) (1994) 601–627.
13. T. Ekman, G. Hedin, Rewritable reference attributed grammars., in: M. Odersky (Ed.), ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings, Vol. 3086 of Lecture Notes in Computer Science, Springer, 2004, pp. 144–169.
14. T. Ekman, G. Hedin, Modular name analysis for Java using JastAdd, in: Post-proceedings of GTTSE 2005, To appear in Lecture Notes in Computer Science, Springer-Verlag, 2006.
15. H. H. Vogt, S. D. Swierstra, M. F. Kuiper, Higher order attribute grammars, in: Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation, ACM Press, 1989, pp. 131–145.
16. J. Saraiva, Purely functional implementation of attribute grammars, Ph.D. thesis, Utrecht University, The Netherlands (1999).
17. E. Van Wyk, O. d. Moor, K. Backhouse, P. Kwiatkowski, Forwarding in attribute grammars for modular language design, in: R. N. Horspool (Ed.), Compiler Construction, 11th International Conference, CC 2002, Grenoble, France, April 8-12, 2002, Vol. 2304 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 128–142.

18. R. Farrow, Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars, in: Proceedings of the SIGPLAN symposium on Compiler contruction, ACM Press, 1986, pp. 85–98.
19. E. Magnusson, G. Hedin, Circular reference attributed grammars - their evaluation and applications., Electr. Notes Theor. Comput. Sci. 82 (3).
20. G. Hedin, E. Magnusson, JastAdd: an aspect-oriented compiler construction system, Science of Computer Programming 47 (1) (2003) 37–58.
21. N. Nystrom, M. R. Clarkson, A. C. Myers, Polyglot: An extensible compiler framework for java, in: Proceedings of 12th International Conference on Compiler Construction, CC 2003, Warsaw, Poland, Vol. 2622 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 138–152.
22. A. Myers, N. Nystrom, X. Qi, Polyglot - A compiler front end framework for building Java language extensions, http://www.cs.cornell.edu/Projects/polyglot/ (2006).
23. GrammaTech, The Synthesizer Generator, http://www.grammatech.com/products/sg/ (2006).
24. J. Grosch, Cocktail - compiler compiler toolkit Karlsruhe, http://www.cocolab.com/en/cocktail.html (2006).
25. A. Sloane, W. M. Waite, U. Kastens, Eli - translator construction made easy, http://eli-project.sourceforge.net/ (2006).
26. Lex Augusteijn, The Elegant Homepage, http://www.research.philips.com/technologies/syst%5fsoftw/elegant/ (2006).
27. M. Kuiper, D. Swierstra, M. Pennings, H. Vogt, J. Saraiva, Lrc: A purely functional, higher-order attribute grammar based system, http://www.di.uminho.pt/ jas/Research/LRC/lrc.html (2006).
28. D. Swierstra, A. Baars, UAG - Utrecht Attribute Grammar System, http://www.cs.uu.nl/wiki/Center/AttributeGrammarSystem (2006).
29. M. van den Brand, P. Klint, The ASF+SDF MetaEnvironment, http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment (2006).
30. E. Visser, M. Bravenboer, R. Vermaas, Stratego: Strategies for Program Transformation, http://www.program-transformation.org/Stratego/WebHome (2006).
31. James R. Cordy, TXL - Source Transformation by Example, http://www.txl.ca (2006).
32. K. Olmos, E. Visser, Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules., in: R. Bodík (Ed.), Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings, Vol. 3443 of Lecture Notes in Computer Science, Springer, 2005, pp. 204–220.

Paper III

Benchmarking the JastAdd Extensible Java Compiler

Torbjörn Ekman and Görel Hedin

Department of Computer Science, Lund University, Sweden
(torbjorn|gorel)@cs.lth.se

Benchmarking the JastAdd Extensible Java Compiler

Torbjörn Ekman and Görel Hedin

Department of Computer Science, Lund University, Sweden
(torbjorn|gorel)@cs.lth.se

Abstract This paper compares the JastAdd Extensible Java Compiler to other Java compilers. The compilers are tested for compliance by using the Jacks test suite. Sample applications of various sizes are compiled to evaluate compilation speed. The source code for the various compilers are then compared in terms of size. The JastAdd Extensible Java Compiler matches the state of the art when it comes to compliance. Its size is less than two-thirds of hand-written compilers, and its compilation time is well within a factor of four compared to the fastest Java based compiler.

1 Introduction

The JastAdd Extensible Java compiler [EH06c] is a full Java 1.4 compiler including compile-time checks and bytecode generation implemented in the Rewritable Reference Attributed Grammars (ReRAGs) specification formalism [EH04]. ReRAGs is a high-level formalism for developing program-processing tools, e.g., compilers and code analyzers, with focus on modularity and extensibility. Previous work shows how to use ReRAGs to support several different decomposition criteria for modularization: as separate computations on the program model, as a base language and language extensions, and the same decomposition as used in a language specification for traceability [EH06a,EH06b]. Several synergistic language mechanisms are combined in one coherent framework to achieve the desired modularization: declarative attributes for automatic scheduling of computations, inheritance for model modularization, inter-type declarations for cross-cutting concerns, and rewrites that allow computations to be expressed on the most suitable model. These mechanisms are further explained in [EH04,EH06d].

Java 1.4 [GJSB00] is a large complex language and implementing a complete compiler for it is a substantial undertaking, both because the language contains many idiosyncrasies that must be handled, and because it is an object-oriented language with many non-trivial constructs. To evaluate our implemen-

tation we compare it to several well-known Java compilers in terms of language compliance, compilation time, and implementation size.

The rest of this paper is structured as follows. Section 2 describes the compilers that we compare our JastAdd based compiler to. The compilers are tested for complience using the Jacks test suite in Section 3. Section 4 compares compilation times and Section 5 compares the sizes of the various implementations. Section 6 concludes the paper.

2 Java compilers background

This section describes the compilers that are used in the following comparison. We have included both industrial strength Java compilers as well as a few research prototypes that target modular compiler implementation. If there exists both a Java 1.4 and a Java 5 version of the compiler we include both. The reason is that we want to use the latest version to compare for compliance but use the Java 1.4 version when comparing implementation size. Since our work targets Java 1.4 it would be unfair to compare our code size to a full Java 5 compiler. However, we run all compilers in Java 1.4 mode during compliance testing, even the Java 5 ones. We will use the shorthand in Figure 1 to name the various compilers in the following comparison.

javac1.4	Sun javac version 1.4.2 (Java 1.4)
javac1.5	Sun javac version 1.5.0 (Java 5)
eclipse1.4	Eclipse 2.1.3 (Java 1.4)
eclipse1.5	Eclipse 3.1.2 (Java 5)
jikes	Jikes 1.22 (Java 1.4)
gcj	GNU Compiler for Java 4.0.2 (Java 1.4)
polyglot	Polyglot 1.3.2 (Java 1.4 front-end)
jaco	JaCo 0.6.1 (Java 1.4)
jastadd	JastAdd 1.0 (Java 1.4)

Figure 1. Compiler abbreviations used in the comparison

Sun javac The standard compiler in the Java Platform, Standard Edition, is named javac [jav06]. It is implemented in Java and the source code is available under either the Sun Community Source License or the Java Research License. We have chosen to include two versions of the compiler, 1.4.2 and 1.5.0. The source version of the 1.4.2 compiler is based on the Generic Java Compiler and

there may therefore be some generics related code even in the 1.4.2 version even though generics are not supported until 1.5.0.

Eclipse compiler The Eclipse project contains an incremental Java compiler [ecl06]. It is based on technology evolved from the VisualAge Java compiler. Even though the compiler is incremental it is possible to run it in batch mode which we have done in all comparisons. We have also removed IDE specific compiler code to allow for a fair comparison to stand alone compilers. Version 2.1.3 of the compiler is included in the comparison since it is the last version to support only Java 1.4. It is licensed under the Common Public License v.1.0. The latest version that also supports Java 5 is 3.1.2, which is available under the Eclipse Public License v1.0.

Jikes Compiler Jikes is a high-performance open-source Java 1.4 compiler written in C++ [jik06]. It was originally developed by IBM at T. J. Watson Research Center but is now maintained by an open source community. The comparison is based on version 1.22 which is hosted by SourceForge under the IBM public License.

The GNU Compiler for Java GCJ is the Java compiler in the GNU Compiler Collection [gcj06]. It is written in C and generates either Java bytecode (class files) or native machine code. We generate bytecode in this comparison and used version 4.0.2 that is licensed under the GNU General Public License.

Polyglot extensible Java front-end Polyglot is a compiler front-end framework for building Java language extensions [NCM03,MNQ06]. It is a Java class library that can be extended through inheritance to create a compiler for a language that is an extension to Java. The implementation uses a novel visitor design that allows for modular extension of both static semantic analysis and language structure. The comparison uses version 1.3.2 which is licensed under the GNU Lesser General Public License. There is no back-end included in the distribution but we include it for its extensibility support. Polyglot has been combined with the Soot framework to form a full compiler [VRHS+99].

JaCo extensible Java compiler JaCo is an extensible compiler for version 1.4 of the Java programming language [jac06b]. The compiler is written in a slightly extended Java dialect called Keris that supports extensible algebraic data types with defaults [ZO01]. There is a compiler for Keris named KeCo

which is implemented as a modular extension to JaCo. The comparison includes version 0.6.1 of the system which is licensed under the Q Public License version 1.0.

3 Compiler compliance

We have used the Jacks test suite to test the compilers for compliance. Jacks is a free test suite designed to detect bugs in a Java compiler [jac06a]. It was originally developed by IBM but is now maintaned by the Mauve project. The suite structure mimics chapters from the Java Language Specification, Second Edition [GJSB00]. It is worth noticing that Jacks tests the static semantic analysis of compilers and is not designed to test a Java runtime (JVM) or Java class libraries.

We ran the full test suite on the compilers described in the previous section and the result is shown in Figure 2. We have turned off warnings in all compilers since they often give hints to common mistakes which are not errors strictly speaking, e.g., a warning that a finally block can not complete normally. The Java 5 compilers (javac1.5 and eclipse1.5) were running in 1.4 compability mode.

Although JastAdd passes more tests than any of the other comnpilers, we do not claim superiority to either compiler but merely use the number of failed test cases as an indication of completeness of the compiler implementation. The skipped tests are errors that cause the compiler to run into a never ending loop. The test that is skipped for javac1.4, javac1.5, eclipse1.4, eclipse1.5 and jastadd is due to a bug in the standard class library. A conversion method that converts a string representation of a floating point number into its binary counterpart fails to terminate. A weakness in the test suite is that it only tests the static semantics of Java and not run-time behavior. It would have been nice to run the more extensive JCK test suite that also tests the generated code, but current licensing prevents us from running that suite [JCK06]. The following compilers in the comparison have licensed and passed the JCK compiler test suite: javac1.4, javac1.5, eclipse1.4, and eclipse1.5.

4 Compilation time

To evaluate the speed of our generated compiler we have compared compilation times for various applications. These applications are described in Figure 3. Each application was compiled five times and the shortest compilation time for each compiler is shown in Figure 4. We have only included the compilation time for an application if the compiler managed to compile the application without false errors.

Compiler	% passed	# passed	# skipped	# failed
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
gcj	*87.3 %*	3919	0	572
polyglot	*90.5 %*	4065	49	377
jaco	*78.0 %*	3505	3	983
jastadd	*99.5 %*	4468	1	22

Figure 2. Results from running the Jacks test suite. Skipped tests are errors that cause the compiler to run into a never ending loop.

We use version 1.4.2 of javac as baseline when comparing compilation times. The results indicate that our generated compiler is less than four times slower than the fastest hand-written java compiler. We also notice that the C/C++ based implementations are significantly faster than the Java based implementations. The compilers that scored low in the compliance test failed to compile one or more real applications. This result indicates that there is at least some correlation between passing the synthetic test suite and real applications.

JUnit Version 3.8.1 of the JUnit testing framework. A small application that all compilers pass, roughly 3.6 kLOC.

JDK A subset of the JDK class library. More than 112 kLOC including the following packages: java.lang, java.util, java.io, java.math, java.net, and java.awt.

JDTComp The Java compiler in Eclipse Java Developement Tools 3.1.2. A substantial stand-alone Java application of 83 kLOC.

Polyglot The Polyglot Java front-end with buldled parser generator JavaCUP consisting of 48 kLOC. Somewhat different coding style with extensive use of visitors, polymorphism, and inheritance.

Figure 3. Applications used to benchmark compilation time. The application sizes are measured using David A. Wheeler's 'SLOCCount'.

Compiler	JUnit (3.6)	JDK (112)	JDTComp (83)	Polyglot (48)
javac1.4	1.06s (*100 %*)	5.05s (*100 %*)	3.44s (*100 %*)	3.34s (*100 %*)
javac1.5	1.38s (*130 %*)	6.07s (*120 %*)	4.41s (*128 %*)	4.48s (*134 %*)
eclipse1.4	1.74s (*164 %*)	6.70s (*133 %*)	5.20s (*151 %*)	4.90s (*147 %*)
eclipse1.5	1.46s (*138 %*)	6.23s (*123 %*)	4.74s (*138 %*)	4.40s (*132 %*)
jikes	0.21s (*20 %*)	1.80s (*36 %*)	1.21s (*35 %*)	1.09s (*33 %*)
gcj	0.39s (*37 %*)	fail	fail	fail
polyglot	3.60s (*340 %*)	fail	fail	49.32s (*1177 %*)
jaco	1.57s (*148 %*)	fail	fail	fail
jastadd	3.17s (*299 %*)	17.23 (*341 %*)	12.95s (*376 %*)	11.7 (*350 %*)

Figure 4. The compilation time for a few applications where the javac1.4 compiler is used as baseline. The application size in thousand lines of code is included in the header. An overview of the applications is shown in Figure 3

5 Implementation size

While the compilers are implemented in different language dialects or even completely different languages it is still interesting to compare implementation sizes to get a rough measure of implementation effort. We have used SLOC-Count by David A. Wheeler to count the number of lines of code in the source files [Whe06]. The tool compensates for differences in coding conventions by removing comments and white-space for a number of languages. Since both JastAdd and Jaco extend Java we use the built-in Java schema for these tools as well as plain Java compilers.

Figure 5 shows the size for each measured compiler. We have included lines of code, total number of tokens, and compressed size for each compiler. The number of tokens and compressed size should should give a hint about the overall implementation entropy. We have used javac1.4 as the baseline here as well. We are somewhat surprised by the large differences in compiler source size. Most compilers are more than twice as large as the base javac1.4 compiler. JastAdd and JaCo stand out being significantly smaller than the base compiler. Both compilers use a Java dialect where the language has been extended to allow for implementation of extensible compilers. The complience result for JaCo indicates that it is incomplete, which makes it hard to draw any conclusions from its implementation size. However, the same complience test indicates that JastAdd is a complete Java 1.4 compiler while being two-thirds the size of the smallest complete handwritten compiler. This shows that the ReRAGs formalism can lead to significantly smaller implementations.

Compiler	# kLOC	# kTokens	# kbyte
javac1.4	21 (*100 %*)	106 (*100 %*)	156 (*100 %*)
javac1.5	30 (*143 %*)	155 (*146 %*)	247 (*158 %*)
eclipse1.4	57 (*271 %*)	288 (*272 %*)	356 (*228 %*)
eclipse1.5	83 (*395 %*)	411 (*388 %*)	508 (*326 %*)
jikes	70 (*333 %*)	342 (*323 %*)	437 (*280 %*)
gcj	63 (*300 %*)	348 (*328 %*)	500 (*321 %*)
polyglot	39 (*186 %*)	220 (*208 %*)	279 (*178 %*)
jaco	16 (*76 %*)	73 (*67 %*)	105 (*82 %*)
jastadd	14 (*67 %*)	58 (*55 %*)	82 (*53 %*)

Figure 5. The source code size of the compilers using javac1.4 as baseline. The kLOC attribute is measured using David A. Wheeler's 'SLOCCount'.

6 Conclusions

We have compared the JastAdd Extensible Java Compiler to other well-known Java compilers. The generated compiler passes at least as many tests as production compilers such as javac from Sun Microsystems, the Eclipse Project Java Compiler, and the Jikes high-speed java compiler. The approach scales and can be used for fairly large programs (> 100 kLOC). The generated compiler is less than four times slower than a handwritten java compiler. The source code for the JastAdd Extensible Java Compiler is only two-thirds the size of the smallest hand-written complete Java compiler.

References

[ecl06] Eclipse Java Compiler, Eclipse Java Development Tools 3.1.2, 2006. http://download.eclipse.org/eclipse/downloads/drops/R-3.1.2-200601181600/.

[EH04] Torbjörn Ekman and Görel Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004*, volume 3086 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[EH06a] Torbjörn Ekman and Görel Hedin. Modular name analysis for Java using JastAdd. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering, Braga, Portugal, July 4–8, 2005*, Lecture Notes in Computer Science. Springer-Verlag, 2006. To appear.

[EH06b] Torbjörn Ekman and Görel Hedin. Pluggable non-null types for Java. Technical report, 2006. Unpublished manuscript, http://jastadd.cs.lth.se.

[EH06c] Torbjörn Ekman and Görel Hedin. The JastAdd compiler compiler system, 2006.
http://jastadd.cs.lth.se.

[EH06d] Torbjörn Ekman and Görel Hedin. The JastAdd System – modular extensible compiler construction. Technical report, 2006. Submitted for publication, http://jastadd.cs.lth.se.

[gcj06] The GNU Compiler for the Java Programming Language, 2006.
http://gcc.gnu.org/java/.

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

[jac06a] The Jacks compiler test suite, the Mauve Project, 2006.
http://sources.redhat.com/mauve/.

[jac06b] JaCo Java Compiler, The Programming Language Keris, 2006.
http://lampwww.epfl.ch/ zenger/keris/.

[jav06] javac in Java 2 Platform, Standard Edition 5.0, 2006.
http://java.sun.com/j2se/1.5/.

[JCK06] JCK test suite - Technology Compability Kit for J2SE, 2006.
https://jck.dev.java.net/.

[jik06] Jikes high performance Java compiler, 1.22, 2006.
http://jikes.sourceforge.net/.

[MNQ06] Andrew Myers, Nathaniel Nystrom, and Xin Qi. Polyglot - A compiler front end framework for building Java language extensions, 2006.
http://www.cs.cornell.edu/Projects/polyglot/.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Proceedings of 12th International Conference on Compiler Construction, CC 2003, Warsaw, Poland*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer-Verlag, 2003.

[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.

[Whe06] David A. Wheeler. Sloccount, 2006. http://www.dwheeler.com/sloccount/.

[ZO01] Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 241–252, New York, NY, USA, 2001. ACM Press.

Paper IV

Modular name analysis for Java using JastAdd

Torbjörn Ekman and Görel Hedin

Department of Computer Science, Lund University, Sweden
(torbjorn|gorel)@cs.lth.se

Modular name analysis for Java using JastAdd

Torbjörn Ekman and Görel Hedin

Department of Computer Science, Lund University, Sweden
`(torbjorn|gorel)@cs.lth.se`

Abstract Name analysis for Java is challenging with its complex visibility rules involving nested scopes, inheritance, qualified access, and syntactic ambiguities. We show how Java name analysis including ambiguities related to names of variables, fields, and packages, can be implemented in a declarative and modular manner using the JastAdd compiler construction system.
Declarative attributes and context-dependent rewrites enable the implementation to be modularized in the same way as the informal Java language specification. The individual rules in the specification transfer directly to equations in the implementation. Rewrites are used to define new concepts in terms of existing concepts in an iterative manner in the same way as the informal language specification. This enables equations to use both context-free and context-dependent concepts and leads to improved separation of concerns. A full Java 1.4 compiler has been implemented to validate the technique.

1 Introduction

The computations done on abstract syntax trees in compilers and related tools are often highly context sensitive. E.g., there are often symbolic names that have different meanings depending on their context. The purpose of name analysis is to bind each name to a declaration and hence resolve the meaning of that name. Name analysis for the Java programming language is challenging with its complex visibility rules involving nested scopes, inheritance, qualified access, and syntactic ambiguities. The purpose of this paper is to show how ambiguities related to names of variables, types, and packages, can be solved in a declarative and modular manner, using the JastAdd compiler construction system.

Consider the qualified name `A.B.C` and the task of binding each individual simple name to its declaration. The meaning depends on the *syntactic context*, e.g., `C` is expected to be a `TypeName` in the **`extends`** clause of a class declaration, and an `ExpressionName` when being the right hand side of an assignment. There are also *contextually ambiguous* names where the set of visible declarations are

required to resolve the name. For example, `A.B` can be the `PackageName` of the top level class `C`, or `A, B, and C` can all be nested `TypeNames`. Such ambiguities should be resolved by reclassification to `TypeNames` if there are visible type declarations and otherwise to `PackageNames`. The Java Language Specification [3] defines the specific rules for visible declarations at each point in a program and how to first classify context-free names according to their syntactic context and then refine them by reclassifying contextually ambiguous names.

JastAdd supports declarative attributes and context-dependent rewrites that enable the implementation to be modularized in the same way as the informal language specification. The individual rules in the specification transfer directly to equations in the implementation. The language specification contains a set of basic language concepts captured by a context-free grammar. There are, however, additional concepts that are context-dependent, e.g., `TypeNames`. Rewrites are used to refine the tree to use not only the basic concepts but also the context-dependent ones. We present a transformational technique to gradually define new concepts in terms of existing concepts, in the same way that they are defined in the informal language specification. This allows for decomposition of complex problems into simpler ones, and it also better supports separation of concerns.

We define a tiny subset of Java named *DemoJavaNames* which captures all the characteristic problems in resolving contextually ambiguous names that occur in full Java. A complete name analysis implementation for DemoJavaNames is presented and included in this paper. We have implemented a full Java 1.4 compiler based on the same technique to verify that the techniques scale to full languages. The system has been validated against the Jacks test-suite and passes more tests than the production quality compilers javac and jikes [1]. While not claiming superiority over either compiler we claim that our implementation is complete while being less then half the size of the handwritten javac compiler.

The rest of this paper is structured as follows. Section 2 introduces the features of JastAdd that are used in the implementation of DemoJavaNames. Section 3 describes the implementation of name lookup, syntactic classification, and reclassification of contextually ambiguous names. Section 4 compares our work to related work and Section 5 concludes the paper.

2 JastAdd Background

The JastAdd compiler construction system combines object-orientation and static aspect-oriented programming with declarative attributes and context-dependent rewrites to allow highly modular specifications. This section gives an

introduction to the JastAdd system, needed to understand the source code listings in Section 3. The evaluation algorithm is described in [7,2] and the system is publically available [1].

2.1 Abstract grammar

The abstract grammar models an object-oriented class hierarchy from which classes are generated that are used as node types in the abstract syntax tree (AST). Consider the grammar in Figure 1. A class is generated for each production in the grammar, e.g., `Prog, CompUnit, ClassDecl`, and may inherit another production by adding a colon followed by the super production, e.g., `LocalVariableDecl : Stmt`.

The right hand side of a production is a list of elements. The default name of an element is the same as its type unless it is explicitly named by prefixing the element with a name and a colon, e.g., the `FieldDecl` has an element named `Type` which is of type `Name`. Elements enclosed in angle brackets are values, e.g., `<name:String>` in `FieldDecl`, while other elements are tree nodes, e.g., `Type:Name` and `Expr` in `Field-Decl`. The tree node element may be suffixed by a star to specify a list of zero or more elements, e.g., `ClassDecl*` in `CompUnit`.

The system generates a constructor and accessor methods for value and tree elements. The accessor method for a value element has the same name as the element, e.g., `String name()`, while the tree element is prefixed by get, e.g., `Name getType()`. List elements have an index to select the appropriate node, e.g., `getClassDecl(int index)`, and there is an accessor for the number of elements in the list, e.g., `int getNumClassDecl()`.

2.2 Declarative attributes

Attribute Grammars [10] have proven useful when describing context-sensitive information for programming languages. Their declarativeness makes it easy to modularize grammars freely, and they integrate well with the object-oriented programming paradigm, in particular when augmented with *reference attributes*, allowing an attribute to be a reference to another tree node object [6]. This section gives a very brief introduction to *synthesized* and *inherited* declarative attributes.

A *synthesized* attribute is similar to a virtual method without side-effects which allows for efficient evaluation using caching. Consider the grammar in Figure 1 and the task to determine whether a `Stmt` node declares a local variable named *name* or not. This can be implemented through a synthesized attribute using the following JastAdd syntax:

```
syn boolean Stmt.isLocalVariableDecl(String name);
eq Stmt.isLocalVariableDecl(String name) {
  return false;
}
eq LocalVariableDecl.isLocalVariableDecl(String name) =
  name().equals(name);
```

Notice that the equation for `LocalVariableDecl` overrides the default equation for its superclass `Stmt`. Notice also the functional styled short-hand for its right-hand side: it uses an expression rather than a block with a return statement. An additional shorthand is possible (but not shown): combining the attribute declaration and the first equation into a single clause by inserting the equation right-hand side before the semicolon in the declaration.

JastAdd supports inter-type declarations [9] where attributes can be added to an existing class in a modular fashion. The target class for each attribute and equation is specified by qualifying its name with the target class name, e.g `Stmt` and `Local-VariableDecl` above. The attribute is then woven into the class hierarchy generated from the abstract grammar.

An *inherited* attribute propagates the context downwards the AST. Consider the task to determine the enclosing `Block` for a `Stmt` node. A block can tell all its enclosed `Stmts` that it is the enclosing `Block` declaration. This can be implemented through an inherited attribute using the following syntax:

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

Equations for inherited attributes are broadcast to an entire subtree in a similar way as for the *including* construct in the Eli attribute grammar system [13]. This subtree is explicitly selected using a child accessor (`getStmt()` in this case). The equation should thus be read as: *define the value for the enclosingBlock attribute in the entire subtree whose root is the node returned by getStmt() in a block node. The value should be* this*, i.e., a reference to the block node defining the equation.*

2.3 Context-dependent rewriting

JastAdd supports declarative context-dependent rewrites to dynamically change the AST. A node of type *S* is automatically rewritten to a node of type *T* when a certain condition is true using the syntax below:

```
rewrite S {
  when(condition())
  to T new T(...);
}
```

The rewrites are context-dependent in that the conditions may depend on synthesized and/or inherited attributes. The rewrites are declarative in that they are performed automatically by a rewrite evaluation engine. In the final tree, no rewrite conditions are true. There may be multiple when-to clauses in which case they are evaluated in lexical order. The evaluation engine is demand-driven and rewrites nodes when they are being visited, interleaved with attribute evaluation. The examples discuss the resulting transformation order for each rewrite as well as interaction with other rewrites and attribute evaluation. The evaluation algorithm is presented in [2].

3 Name analysis for DemoJavaNames

This section presents the implementation of name analysis for a tiny subset of Java that only includes compilation units, packages, nested classes with inheritance, fields, initializers, blocks, local variables, and names. We call this subset *DemoJavaNames* and, while being far from useful as a practical language, it captures all the characteristic problems in resolving contextually ambiguous names that occur in full Java.

The input of the name analysis is a context-free tree constructed by the parser. The result is an attributed tree where all names have been resolved to appropriate name kinds, and have reference attributes denoting the appropriate declaration node. The purpose of the paper is to show how ambiguities related to names of variables, types, and packages, can be solved in a declarative and modular manner, using JastAdd. We will show how each of the rules in the language maps to a specific equation in the attribute grammar.

DemoJavaNames keeps just enough language constructs to illustrate the following name related concepts: multiple kinds of nested scopes, object-oriented inheritance, qualified names, shadowing and hiding, and multiple kinds of variables. To simplify the example we removed all language concepts unrelated to names and we also removed language concepts that duplicate name analysis problems, e.g., we only use classes and not interfaces. For brevity, we also removed some language constructs that do affect name binding, i.e., imports of types and access control. While they are not included in the program listings we discuss how the implementation can be extended to handle these features as well.

Figure 1 presents the abstract grammar for DemoJavaNames. The `Dot` production that represents a qualified name requires further explanation. The parser is expected to build right recursive trees where the `Left` child is always a simple name while the `Right` child may be a `Dot` or a simple name. It is also worth noticing that the names in the grammar are context-sensitive, e.g.,

```
ast Prog ::= CompUnit*;
ast CompUnit ::= <packageName:String> ClassDecl*;
ast ClassDecl ::= <name:String> Super:Name BodyDecl*;

ast abstract BodyDecl;
ast FieldDecl : BodyDecl ::= FieldType:Name
                             <name:String> Expr;
ast MemberClassDecl : BodyDecl ::= ClassDecl;
ast Initializer : BodyDecl ::= Block;

ast abstract Stmt;
ast Block : Stmt ::= Stmt*;
ast LocalVariableDecl: Stmt ::= VarType:Name
                                <name:String> Expr;
ast abstract Expr;
ast abstract Name : Expr ::= <name:String>;
ast Dot : Name ::= Left:Name Right:Name;
ast ExpressionName : Name;
ast PackageName : Name;
ast TypeName : Name;
```

Figure 1. DemoJavaNames abstract grammar. A minimal subset of Java used to illustrate the problems in resolving contextually ambiguous names.

`ExpressionName`, `TypeName`, `Package`. We introduce context-free names and transformations into context-sensitive names in Section 3.3.

The type of names and variable declarations is needed to define qualified lookups and inherited members in later modules. We therefore define the type as an attribute of expressions and declarations. Figure 2 implements the type attribute as a reference to the appropriate declaration. To simplify equations in name binding modules we use a null object to represent unknown types. That way it is always possible to query an expression for members instead of handling the special case where the type is unknown.

The following sections present modules for name lookup and reclassification of ambiguous names followed by a discussion on how to extend the implementation to handle full Java.

3.1 Visible declarations

The most important contextual information used in name analysis is the set of visible declarations at each point in a program. Those declarations are then used to bind names in an actual context to their appropriate declarations. The

```
syn ClassDecl Expr.type() = unknownType();
eq Dot.type() = getRight().type();
eq ExpressionName.type() = lookupVariable(name())!=null ?
    lookupVariable(name()).type() : unknownType();
eq TypeName.type() = lookupType(name()) != null ?
    lookupType(name()) : unknownType();
syn ClassDecl LocalVariableDecl.type() =
    getVarType().type();
syn ClassDecl FieldDecl.type() = getFieldType().type();
```

Figure 2. Type binding for DemoJavaNames where each expression and variable declaration is bound to a class declaration. A null object is used for unknown types to allow for a unified member lookup.

name binding module in Figure 3 defines the `lookupVariable(String name)` attribute in `Name` that provides a binding through a reference to a named visible variable-declaration.

Language constructs that change the set of visible declarations, e.g., introduce new declarations or limit scope for an existing declaration, need to provide an equation for the lookup attribute. DemoJavaNames has two kinds of variables, `LocalVariableDeclarations` declared in `Blocks`, and `FieldDeclarations` declared in `ClassDecls`. The equations for lookup need thus be placed in the `Block` and `ClassDecl` types.

Nested scopes with shadowing The scope of a declaration is the region of the program in which the declaration can be referred to using a simple name. The scope of a declaration often involves nested language elements where declarations in one element are in scope in enclosed elements as well. A declaration may be shadowed in part of its scope by another declaration of the same name.

Both classes and blocks are allowed to be nested in DemoJavaNames and both implement shadowing as well. In Figure 3 the delegation to enclosing context, marked with ②, implements nested scopes. The eager return at first match, marked with ①, implements shadowing.

Declarations in a block have a *declare before use* policy. This is implemented by limiting the range of the block that is searched for declarations at ③. The equation is parameterized by the `index` of the `Stmt` in the element list and the search stops at the `Stmt` that encloses the name.

Inheritance The member fields of a class are not only the locally declared fields but also fields inherited from the superclass. A field is inherited if there

```
  // visible variable or null
  inh Variable Name.lookupVariable(String name);
  // local variables in blocks
  eq Block.getStmt(int index).lookupVariable(String name) {
③   for(int i = 0; i < index; i++)
      if(getStmt(i).isLocalVariableDecl(name))
①       return (LocalVariableDecl)getStmt(i);
②   return lookupVariable(name);
  }
  syn boolean Stmt.isLocalVariableDecl(String name) = false;
  eq LocalVariableDecl.isLocalVariableDecl(String name) =
    name().equals(name);
  inh Variable Block.lookupVariable(String name);
  // member fields in classes
  eq ClassDecl.getBodyDecl().lookupVariable(String name) {
    if(memberField(name) != null)
①     return memberField(name);
②   return lookupVariable(name);
  }
  // members including inheritance
  syn FieldDecl ClassDecl.memberField(String name) {
    for(int i = 0; i < getNumBodyDecl(); i++)
      if(getBodyDecl(i).isField(name))
④       return (FieldDecl)getBodyDecl(i);
⑤   if(getSuper().type().memberField(name) != null)
      return getSuper().type().memberField(name);
    return null;
  }
  syn boolean BodyDecl.isField(String name) = false;
  eq FieldDecl.isField(String name) = name().equals(name);
  inh Variable ClassDecl.lookupVariable(String name);
  // no more nested declarations
  eq Prog.getCompUnit().lookupVariable(String name) = null;
  // abstraction for FieldDecl and LocalVariableDecl
  interface Variable {
    String name();
    ClassDecl type();
  }
  FieldDecl implements Variable;
  LocalVariableDecl implements Variable;
```

Figure 3. Variable binding for DemoJavaNames. Shadowing is implemented by eager return statements marked ①. Nesting is implemented using delegation marked ②. Declare before use is implemented by limiting variable search to the current node index in ③.

is not a local field declaration that hides the field in the superclass. The eager return at ④ implements hiding and the delegation to the superclass at ⑤ implements inheritance.

Canonical type lookup The lookup of visible class declarations is implemented in a similar fashion to variable lookup. The main difference is how the lookup is handled at the compilation unit level. If the type is not found in the current compilation unit then the top level types in compilation units belonging to the same package are considered. This is implemented in Figure 4 by delegation ① to a canonical lookup that takes both the package name and type name into account ②. Inheritance of member classes is implemented in the same way as for variables.

3.2 Qualified lookup

The set of visible declarations for a qualified name depends on the target of the resolved name to the left of the dot. A valid `ExpressionName` can be preceded by either a `TypeName` or an `ExpressionName`. Either way, the `ExpressionName` refers to a member field in the `ClassDecl` that represents the type of the preceding expression. Figure 5 extends the name binding module with qualified lookup. The equation at ① defines the variable lookup to search the `ClassDecl` (that the qualifier's type is bound to) for members.

The lookup attribute is an inherited attribute and thus defined by an equation in an ancestor node. The qualifier to the left of the dot in a qualified name should provide the equation for the name on the right hand side of the dot. This is done by the common ancestor `Dot` which propagates the value of the equation from left to right for variables at ① and types at ②, overriding the lookup defined by an ancestor further up in the AST.

A valid `TypeName` can be preceded by either a `PackageName` or a `TypeName`. If the qualifier is a `PackageName` then the qualified name is the canonical name of the type. But if the qualifier is a `TypeName` then the name refers to a member type. There are thus different rules for the lookup depending on the kind of expression that precedes the name. The `Dot` therefore delegates the lookup to the expression at ② and searches for member types at ③ as the default strategy for expressions while the `PackageName` overrides the lookup at ④ to use canonical type names.

3.3 Determine the meaning of names

The abstract syntax defined so far contains name nodes that are highly context sensitive and can thus not be built by a context-free parser. We now extend

```
// visible type or null object
inh ClassDecl Name.lookupType(String name);

// top level types in compilation unit
eq CompUnit.getClassDecl().lookupType(String name) {
  if(topLevelType(name) != null)
    return topLevelType(name);
  // declarations in same package
① return lookupCanonical(packageName(), name);
}
syn ClassDecl CompUnit.topLevelType(String name) {
  for(int i = 0; i < getNumClassDecl(); i++)
    if(getClassDecl(i).name().equals(name))
      return getClassDecl(i);
  return null;
}
// lookup a type using its canonical name
inh ClassDecl Name.lookupCanonical(String p, String t);
eq Prog.getCompUnit().lookupCanonical(String p, String t)
{
  for(int i = 0; i < getNumCompUnit(); i++)
②   if(getCompUnit(i).packageName().equals(p) &&
        getCompUnit(i).topLevelType(t) != null)
      return getCompUnit(i).topLevelType(t);
  return null;
}
// member classes in class declaration
// analoguous to the member fields implementation
eq ClassDecl.getBodyDecl().lookupType(String name) {
  ... }
// no more nested declarations
eq Prog.getCompUnit().lookupType(String name) = null;
```

Figure 4. Type lookup for DemoJavaNames.

the abstract syntax with additional context-free name nodes that are used for gradually refining the names to reflect their semantic meaning.

The parser constructs unqualified name nodes only using the node type `ParseName`. These nodes are then refined by the name analysis to the resulting nodes listed in Figure 1. To simplify this computation, some of the refinements are done in intermediate steps, making use of two additional node types: `PackageOrTypeName` and `AmbiguousName`, see Figure 6.

```
  eq Dot.getRight().lookupVariable(String name) =
①   getLeft().type().memberField(name);

  eq Dot.getRight().lookupType(String name) =
②   getLeft().qualifiedLookupType(name);

  syn ClassDecl Expr.qualifiedLookupType(String name) =
③   type().memberClass(name);
  eq PackageName.qualifiedLookupType(String typeName) =
④   lookupCanonical(name(), typeName);
```

Figure 5. Qualified lookup of types and fields.

Syntactic classification of names The first step in resolving names is to reclassify the `ParseName` nodes based on their immediate syntactic context. This way some nodes can be directly refined to their final class: `PackageName`, `TypeName`, or `ExpressionName`. However, for some names, the immediate syntactic context is not sufficient, in which case the `ParseName` is refined to `PackageOrTypeName` (for names that must refer packages or types), or `AmbiguousName` (for names where the kind cannot yet be determined at all).

The Java language specification defines the classification process by describing a context and the expected name kind. For instance, a name is syntactically classified as a `TypeName` in the **`extends`** clause of a class declaration. We therefore introduce an inherited attribute `kind()` that describes the syntactic classification in a certain context by referring to an element in an enumeration of the above name kinds. Figure 6 shows the `kind()` attribute declaration at ②, the enumeration at ⑥, and the sample classification description at ③.

A qualifier in a qualified name may depend on the classification of the name it qualifies. For instance, a name is syntactically classified as a `PackageOrTypeName` to the left of the dot in a qualified `TypeName`. However, we still have the same requirement for equations in the ancestor as for qualified names. We therefore introduce another attribute `predKind()` which is delegated from right to left at ④ and the equation corresponding to the above example at ⑤.

The equations for `kind()` and `predKind()` complete the description of classification context and the transformation is almost trivial. The conditional rewrite at ① transforms a `ParseName` node into its syntactically classified counterpart. It is worth noticing that the dependences introduced by the `kind()` attribute equations in combination with demand driven rewriting causes qualified names to be classified from right to left.

```
  ast ParseName : Name;
  ast PackageOrTypeName : Name;
  ast AmbiguousName : Name;

① rewrite ParseName {
    when(kind() == Kind.PACKAGE_NAME)
    to Name new PackageName(name());
    when(kind() == Kind.TYPE_NAME)
    to Name new TypeName(name());
    when(kind() == Kind.EXPRESSION_NAME)
    to Name new ExpressionName(name());
    when(kind() == Kind.PACKAGE_OR_TYPE_NAME)
    to Name new PackageOrTypeName(name());
    when(kind() == Kind.AMBIGUOUS_NAME)
    to Name new AmbiguousName(name());
  }

② inh Kind ParseName.kind();
  eq Prog.getCompUnit().kind() = Kind.AMBIGUOUS_NAME;
③ eq ClassDecl.getSuper().kind() = Kind.PACKAGE_NAME;
  eq FieldDecl.getFieldType().kind() = Kind.TYPE_NAME;
  eq FieldDecl.getExpr().kind() = Kind.EXPRESSION_NAME;
  eq LocalVariableDecl.getVarType().kind() = Kind.TYPE_NAME;
  eq LocalVariableDecl.getExpr().kind() = Kind.EXPRESSION_NAME;

  // propagate information from right to left
④ eq Dot.getLeft().kind() = getRight().predKind();
  syn Kind Name.predKind() = Kind.AMBIGUOUS_NAME;
  eq Dot.predKind() = getLeft().predKind();

  eq PackageName.predKind() = Kind.PACKAGE_NAME;
⑤ eq TypeName.predKind() = Kind.PACKAGE_OR_TYPE_NAME;
  eq ExpressionName.predKind() = Kind.AMBIGUOUS_NAME;
  eq PackageOrTypeName.predKind() = Kind.PACKAGE_OR_TYPE_NAME;
  eq AmbiguousName.predKind() = Kind.AMBIGUOUS_NAME;

⑥ class Kind {
    static Kind PACKAGE_NAME = new Kind();
    static Kind TYPE_NAME = new Kind();
    static Kind EXPRESSION_NAME = new Kind();
    static Kind PACKAGE_OR_TYPE_NAME = new Kind();
    static Kind AMBIGUOUS_NAME = new Kind();
  }
```

Figure 6. Syntactic classification of names depending on their context. The context-free ParseName names are classified and rewritten to any of the five name kinds defined in Kind.

Reclassification of contextually ambiguous names The next step is to reclassify contextually ambiguous names, i.e., `AmbiguousName` and `PackageOrTypeName`, in the context of visible declarations. An `AmbiguousName` is reclassified as an `ExpressionName` if there is a visible variable declaration with the same name. Otherwise, as a `TypeName` if there is a visible type declaration with the same name. Otherwise, as a `PackageName` if there is a visible package with the same name. The corresponding implementation is shown in Figure 7.

```
rewrite AmbiguousName {
  when(lookupVariable(name()) != null)
  to Name new ExpressionName(name());
  when(lookupType(name()) != null)
  to Name new TypeName(name());
  when(hasPackage(name()))
  to Name new PackageName(name());
}
rewrite PackageOrTypeName {
  when(lookupType(name()) != null)
  to Name new TypeName(name());
  when(hasPackage(name()))
  to Name new PackageName(name());
}
inh boolean Name.hasPackage(String name);
eq Program.getCompUnit().hasPackage(String name) {
  for(int i = 0; i < getNumCompUnit(); i++)
    if(getCompUnit(i).packageName().equals(name))
      return true;
  return false;
}
eq Dot.getRight().hasPackage(String name) =
      getLeft().qualifiedHasPackage(name);
syn boolean Expr.qualifiedHasPackage(String name) = false;
eq PackageName.qualifiedHasPackage(String name) =
      hasPackage(name() + '.' + name);
```

Figure 7. Reclassification of Contextually Ambiguous Names.

A contextually ambiguous name is resolved by binding it in the context of its qualifier. There is thus a dependence that the qualifier must be resolved before its right hand side can be resolved. We implement this dependence by making sure that all rewrite conditions in Figure 7 are false when the qualifier

of a name is ambiguous. These conditions are false when there are no visible names. The type of an ambiguous name is `unknownType()` which has no visible member fields or types. To make the property hold we add an attribute `hasPackage(String name)` that is true when there is a visible package with that name and no ambiguous qualifiers. A qualified name `a.b.c` is thus first syntactically classified from right to left because of the dependences in the `kind()` attribute, and then reclassified from left to right.

3.4 Extensions to handle full Java

The DemoJavaNames language lacks some important name-related language constructs available in Java. This section describes the needed changes to the implementation to support full Java.

The implementation can be extended with more nested scopes by providing a new equation for the lookup attribute in each new scope. The various nested scopes are totally decoupled from each other using inherited attributes with parameters. The only constraint is that a scope is nested in another scope if they are on the same path to the root node. A `ForStmt` may for instance provide an equation (very similar to the equation for `Block` in Figure 3) that searches for `LocalVariableDeclarations` in its init-clause. Type imports extend the scope of type declarations and can be implemented by inserting a search for matching imports at ① in Figure 4. Java 5 [4] constructs such as *static imports* and the *enhanced for statement* can be supported using the same technique by adding a search for imported fields in the `CompUnit` node type and a lookup equation for local variable declarations in the enhanced for statement AST node.

Java supports access control where modifiers impose visibility constraints on names. Access control limits inheritance in that only non private accessible members are inherited from the superclass. This is easily implemented by adding a filter at ⑤ in Figure 3 that removes private non accessible fields. Access control also affects qualified lookups. The type of a qualifier must for instance be accessible and there are also additional constraints when the qualifier is an `ExpressionName`. Such behavior can be implemented by filters at ① and ② in Figure 5. The specialized rules for `ExpressionName` may require the qualified lookup for fields to be extended to the variant used for types. The filter can then be placed on the `ExpressionName` qualifier.

DemoJavaNames supports inheritance from classes only while Java also supports interfaces. Interfaces complicate name analysis somewhat in that multiple inheritance may cause several fields with the same name to be inherited. This is only an error if a name refers to the ambiguous fields and the error detection can thus not occur in the `ClassDecl` directly but needs to be deferred to

a `Name` node. This can be implemented by turning the lookup attribute into a set of references instead of a single reference. This does not affect the described modularization, but a few equations need to be changed to handle sets. Lookup equations defined to reference a single declaration are changed to a set of declarations, e.g., `eq Block.getStmt(int index).lookupVariable(String name)` in Figure 3 should return a set with a single reference to a variable declaration. Equations that expect a single reference need to ensure that the queried set contains a single reference and then extract that reference, e.g., Figure 2 defines `eq ExpressionName.type()` that should extract a single type-declaration reference or return `unknownType()`. If a name binds to more than one element the name is ambiguous and a compile-time error is reported.

4 Related work

Transformation technology is commonly used in compiler construction to refine the AST to include context-sensitive information for later passes. Our approach differs from similar techniques in the use of context-dependent rewrites interleaved with attribute computations. Rewrites allow us to gradually define new concepts in terms of existing concepts, in the same way commonly used in informal language definitions. The fine-grained interaction between attribute computation and rewriting enables the immediate use of these concepts in equations without the need of defining separate passes. This is a key mechanism that allows complex analysis problems like Java name resolution to be broken down into small simple steps. To our knowledge, there are no other systems supporting similar mechanisms. Higher-order attribute grammars [17,12] allow the AST to be used as the only data structure, and combined with forwarding [15] it may be possible to use in a similar fashion, but as far as we know, forwarding has only been implemented in prototypes built on top of Haskell, and it is unclear how the practical performance would scale to full languages like Java.

The basic idea of name analysis for object-oriented languages based on explicit name bindings was used by ourselves earlier for simpler object-oriented languages [5], [6], and by Vorthmann in his visibility graph technique [18]. Vorthmann also uses a filtering technique to take care of constructs that limit declaration visibility. However, these approaches did not use context-dependent node types, which contribute substantially to making the approach modular. There is some other work aiming at separating the name analysis from other phases of a compiler, most notably the work on Kastens and Waite on an abstract data type for symbol tables [8]. The current version of Eli [13] contains an extensible library of modules for a large variety of scope rules, e.g., single inheritance, multiple inheritance, declare before use.

JastAdd lets context-dependent computations drive the transformations but it is interesting to compare to the opposite approach: letting transformations drive contextual computations commonly used in transformation systems such as ASF+SDF [14] and Stratego [16]. An important difference is that transformation systems typically handle contextual information by using an external database that is updated during the transformations. This requires the user to explicitly associate database updates with particular transformation rules or phases. The traversal order must thus take contextual dependences, which can be highly nonlocal, into account. In contrast, JastAdd uses the contextual dependences to derive a suitable traversal strategy. The Stratego system has a mechanism for dependent dynamic transformation rules [11], supporting certain context-dependent transformations, but it is not clear how this could be used for implementing name binding and similar problems in object-oriented languages.

5 Conclusions

We have presented a technique to implement name analysis for the Java programming language. The main contribution of the paper is to show how complex problems in name analysis including ambiguities related to names of variables, types, and packages can be solved in a declarative and modular way. The use of declarative attributes and contextual rewrites allow the implementation to be modularized in the same way as the language specification. Context-free as well as context dependent concepts in the language can be used directly in attributes and equations. It is worth noticing that the implementation can be freely modularized according to different criteria. A language extender may for instance choose to define a module with all attributes and equations related to a new language construct. The granularity of what can be modularized is a single attribute or equation, thereby providing excellent support for separation of concerns.

We have defined a small subset of Java that captures all the characteristic problems in resolving contextually ambiguous names. The implementation using JastAdd is less than 200 lines of code, most of it included in the paper. The source code and the JastAdd tool are available for download at [1]. The technique has been used to implement a full Java 1.4 compiler to verify that the technique scales to the full language. The system has been validated against the Jacks test-suite and passes more tests than the production quality compilers javac and jikes [1] while being roughly half the size of the handwritten javac compiler.

Acknowledgements

We are grateful to Calle Lejdfors and the anonymous reviewers for valuable feedback and helpful comments.

References

1. T. Ekman and G. Hedin. The JastAdd II compiler compiler system. http://jastadd.cs.lth.se.
2. T. Ekman and G. Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004*, volume 3086 of *LNCS*. Springer-Verlag, 2004.
3. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.
4. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Third Edition*. Addison-Wesley, Boston, Mass., 2005.
5. G. Hedin. An overview of door attribute grammars. In *Proceedings of Compiler Construction 1994*, volume 786 of *LNCS*, pages 31–51. Springer-Verlag, 1994.
6. G. Hedin. Reference attribute grammars. In *Informatica (Slovenia)*, 24(3), 2000.
7. G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.
8. U. Kastens and W. M. Waite. An abstract data type for name analysis. *Acta Informatica*, 28(6):539–558, 1991.
9. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001*, volume 2072 of *LNCS*, pages 327–355. Springer-Verlag, 2001.
10. D. 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).
11. K. Olmos and E. Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In *Proceedings of Compiler Construction 2005*, volume 3443 of *LNCS*. Springer-Verlag, 2005.
12. J. Saraiva. *Purely functional implementation of attribute grammars*. PhD thesis, Utrecht University, The Netherlands, 1999.
13. A. Sloane, W. M. Waite, and U. Kastens. Eli - translator construction made easy. http://eli-project.sourceforge.net/.
14. M. van den Brand and P. Klint. The ASF+SDF MetaEnvironment. http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment.
15. E. Van Wyk, O. d. Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proceedings of Compiler Construction 2002*, volume 2304 of *LNCS*, pages 128–142. Springer-Verlag, 2002.
16. E. Visser, M. Bravenboer, and R. Vermaas. Stratego: Strategies for Program Transformation. http://www.program-transformation.org/Stratego/WebHome.
17. H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation*, pages 131–145. ACM Press, 1989.

18. S. A. Vorthmann. Modelling and specifying name visibility and binding semantics. Technical Report CMU//CS-93-158, 1993.

Paper V

Pluggable non-null types for Java

Torbjörn Ekman and Görel Hedin

Department of Computer Science, Lund University, Sweden
(torbjorn|gorel)@cs.lth.se

Pluggable non-null types for Java

Torbjörn Ekman and Görel Hedin

Department of Computer Science, Lund University, Sweden
(torbjorn|gorel)@cs.lth.se

Abstract Static type systems allow for early detection of errors and enable developers to clearly document their intent. This paper shows how the existing type system of Java can be extended in a modular fashion while reusing existing compiler infrastructure. This allows for pluggable type systems that can be enabled at will.
Non-null types is a type-based approach to statically detect possible null pointer violations in code. Fähndrich and Leino showed how an object-oriented language such as Java or C# could be extended with non-null types and also implemented a prototype for C#.
We have extended the JastAdd Extensible Java Compiler, in a modular way, to include non-null types. We also extend previous work by presenting a type refinement algorithm that retrofits legacy code to include non-null types. The algorithm has been used to infer non-null annotations in the JDK standard class library. Both implementations are compact, the non-null extension being around 220 lines of code while the type refinement algorithm is less than 460 lines of code.

1 Introduction

Static typing allows for early detection of certain classes of errors and allows developers to clearly document their intent in the form of type signatures. In this paper we show how type system extensions to Java can be implemented in a compact and modular way, by extending the JastAdd Extensible Java 1.4 Compiler [EH], using ReCRAGs (Rewritable Circular Reference Attributed Grammars) [EH04,MH03]. This modularity is important from several perspectives. First, it makes type system additions possible to implement with moderate effort, allowing reuse of existing compiler infrastructure. Second, it opens up for so called pluggable types [Bra04], where an additional type system is run optionally to statically detect additional errors in code. The base language (Java) is not changed as such: Any program that is accepted by our extended compiler will also be accepted by a standard Java compiler (ignoring the annotations), and the runtime behavior is the same.

Non-null types are available in some newer object-oriented languages, e.g., Spec# [BLS04] and Nice [Bon]. The idea is to let the compiler detect that

certain expressions will never have the value null. These expressions can be safely dereferenced without any risk of leading to null pointer exceptions at runtime. To help the compiler in this analysis, the source code can be annotated using modifiers on reference declarations. Fähndrich and Leino [FL03] showed how an object-oriented language such as Java or C# could be extended with non-null types and they also implemented a prototype for C#.

We use non-null types for Java as an example of how to extend the type system in a modular fashion. A small kernel language capturing characteristic type-checking properties of Java is designed, and we show how that language can be extended with non-null types. The same technique is used in a full Java 1.4 implementation that is publically available.

When using this language extension a problem is that legacy code is not annotated. Experiments by Fähndrich and Leino showed that if the default case is that references are not allowed to be null, quite few annotations are needed in practice. However, combined with legacy code the reverse default rule must be used: references may possibly be null, e.g., legacy methods may return null values. The annotated new code must always assume null values from legacy code, and add extra checks against null when using such code in order to be safe from possible run-time null pointer exceptions.

We suggest an improved approach by inferring annotations in legacy code and thereby refine existing type declarations. The idea is to add annotations to a library of legacy code, e.g., the JDK, to get a safe conservative approximation of which references in the legacy code are always non-null. These inferred annotations can then be used by the explicitly annotated code to be able to safely use much of the legacy code without extra guarding null-checks. The type refinement analysis over the legacy code is a whole program analysis since it is necessary to take inheritance and overriding into account in order to obtain good approximations that are useful (i.e., a safe but uninteresting approximation would be to infer that all references in the library code are possibly null).

The rest of this paper is structured as follows. Sections 2 and 3 present the necessary background to non-null types and the JastAdd system to understand the presented implementation techniques. Section 4 defines a base language which is extended with non-null types in Section 5. Section 6 discusses extensions to cover full Java 1.4, and how the technique can be used for other type extensions. The type refinement algorithm and its implementation is presented in Section 7. We evaluate the presented techniques in Section 8. Related work is discussed in Section 9 and we conclude the paper and discuss future work in Section 10.

2 Non-null types background

The non-null types extension is based on the work on non-null types for object-oriented languages, as presented by Fähndrich and Leino [FL03]. Their approach differs from earlier work on non-null types by taking inheritance and object initialization into account. Fähndrich and Leino made a non-modular prototype implementation for C# while we present a modular extension to Java and focus on implementation techniques. We also extend their approach by adding a simple but effective inference algorithm in order to handle legacy code that does not have non-null annotations.

The purpose of non-null types is to add the possibility to distinguish *non-null* references from *possibly-null* references in the type system. This enables the compiler to statically detect null-related errors at compile-time. The programmer already needs to consider whether a value may be null or not and the special handling of null values is error prone. It is clearer to make this design explicit in the code, and making use of these type invariants in order to write simpler and safer code. In particular, conditionals guarding against null can safely be removed for non-null references, and the compiler can give warnings or errors if there are missing null guards for expressions that are possibly null.

We therefore split reference types into possibly-null and non-null types. For each possibly-null type T there is a non-null counterpart T^-. Where the language requires an expression of reference type T but stipulates that the VM throws a null pointer exception when the expression evaluates to null, we instead require an expression of reference type T^-.

The programmer typically annotates the type names of reference declarations in order to specify a non-nullness property. The subtype relation is extended to include these new types. Consider the example in Figure 1. A newly created instance is clearly non-null and thus allowed to be assigned to a non-null typed variable. A possibly-null typed variable may be assigned an expression typed by its non-null counterpart, i.e., T^- is a subtype of T. A conditional statement that checks a possibly-null variable against *null*, automatically casts the variable into its non-null counterpart, as long as it is not assigned a possibly-null value in that branch. This is a conservative approximation that could be improved by taking control-flow into account. However, the simple approach described above works well in practice. A qualified variable or method-name must be qualified by a value of a non-null type, otherwise a possible null pointer violation is reported.

The (reference) type hierarchy can be modelled by the type lattice shown in Figure 2. A type T is a subtype of another type S if there is a path in the upward direction from T to S. The subtype relation is extended to include the new types using the rules in Figure 2.

```
T- t = new T(); // allocate a non-null object
T n = t; // this direction is allowed
if(n != null) {
  t = n; //n is of type T- in this context
}
int x = t.f; // type of t must be non-null
```

Figure 1. Example of non-null types

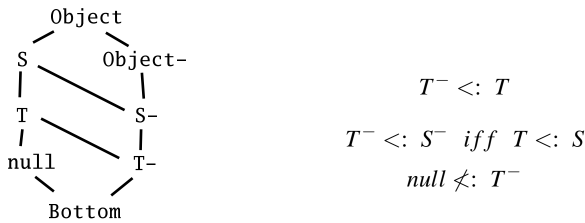


Figure 2. Type lattice and extended subtype relation including non-null types.

The combination of non-null instance fields and object initialization complicates matters. Instance fields are not always initialized at declaration, but often in a constructor body or instance initializer. To support such initializations, we allow instance fields to be declared non-null as long as they are definitely assigned at the end of each constructor. This guarantees that the variable has a non-null value after the object has been initialized. However, the reference to a newly constructed, not yet initialized object, is accessible through the **`this`** reference and may thus expose instance fields that may not yet have been initialized. Consider the listings in Figure 3. The left hand is actually equivalent to the right hand side from a code generation point of view. There is an implicit call to the super constructor and fields are not initialized until after that call is completed. The constructor in `A` is thus reached before `t` has been initialized and the virtual call to `print()` reaches the implementation in `B` that uses the `t` field prior to its initialization.

The problem is that **`this`** within a constructor references a partially initialized object. A type based solution to this problem is to introduce references to partially initialized objects. The type system is extended with not only non-null references but also with references to uninitialized objects. These references are called *raw* and when reading fields in raw objects we expect them to be possibly-null regardless of their annotations. Since instance methods do not

```
class A {
  String s = ''s'';
  A() {
    print();
  }


  void print() {
    System.out.println(s);
  }
}
class B extends A {
  String t = ''t'';
  B() {
  }


  void print() {
    System.out.println(s+t);
  }
}
```

```
class A extends Object {
  String s;
  A() {
    super();
    this.s = ''s'';
    this.print();
  }
  void print() {
    System.out.println(this.s);
  }
}
class B extends A {
  String t;
  B() {
    super();
    this.t = ''t'';
  }
  void print() {
    System.out.println(
      this.s + this.t);
  }
}
```

Figure 3. Access to fields in partially initialized objects. The generated code is identical for both examples.

have an explicit declaration of the **this** reference, the entire method may be annotated as being *raw* which implies that the type of **this** is its raw counterpart in that method body.

The technique can be improved by further extending the type system to gradually refine the rawness upto a class in the class hierarchy that is guaranteed to have been initialized. See [FL03] for further details on *raw upto*.

3 JastAdd Background

The JastAdd compiler construction system combines object-orientation and intertype declarations with declarative attributes and context-dependent rewrites to allow highly modular specifications. Behavior may be specified declaratively using the Rewritable Circular Reference Attributed Grammars (ReCRAGs) formalism [EH04,MH03] or imperatively using Java code. This section gives an introduction to the JastAdd system, needed to understand the source code list-

ings in this paper. The evaluation algorithm is described in [HM03,EH04,MH03] and the system is publically available at http://jastadd.cs.lth.se

3.1 Abstract grammar

The abstract grammar models an object-oriented class hierarchy from which classes are generated that are used as node types in the abstract syntax tree (AST). Consider the grammar in Figure 4 in Section 4.1. A class is generated for each production in the grammar, e.g., `Program, TypeDecl, Expr`, and may inherit another production by adding a colon followed by the super production, e.g., `ClassDecl : TypeDecl`.

The right hand side of a production is a list of elements. The default name of an element is the same as its type unless it is explicitly named by prefixing the element with a name and a colon. E.g., the `TypeDecl` has an element named `Extends` which is of type `TypeName`. Elements enclosed in angle brackets are values, e.g., `<Name:String>` in `TypeDecl`, while other elements are tree nodes, e.g., `Extends:TypeName` and `BodyDecl` in `TypeDecl`. The tree node element may be suffixed by a star to specify a list of zero or more elements, e.g., `BodyDecl*` in `TypeDecl`.

The system generates a constructor and accessor methods for value and tree elements. The accessor method is prefixed by `get`, e.g., `TypeName getExtends()`. List elements can be queried for the number of elements in the list using `int getNumBodyDecl()`. Elements are selected using an index to specify the appropriate child through `getBodyDecl(int index)`,

3.2 Declarative attributes

Attribute Grammars [Knu68] have proven useful when describing context-sensitive information for programming languages. Their declarativeness makes it easy to modularize grammars freely, and they integrate well with the object-oriented programming paradigm, in particular when augmented with *reference attributes*, allowing an attribute to be a reference to another tree node object [Hed00]. This section gives a very brief introduction to *synthesized* and *inherited* declarative attributes.

A *synthesized* attribute is similar to a virtual method without side-effects which allows for efficient evaluation using caching. Consider the grammar in Figure 4 in Section 4.1 and the task to determine whether an `Expr` node accesses a field named *name* or not. This can be implemented through a synthesized attribute using the following JastAdd syntax:

```
syn boolean Expr.isFieldName(String name);
eq Expr.isFieldName(String name) { return false; }
```

```
eq FieldName.isFieldName(String name) =
      getName().equals(name);
```

Notice that the equation for `FieldName` overrides the default equation for its superclass `Expr`. Notice also the functional styled short-hand for its right-hand side: it uses an expression rather than a block with a return statement. An additional shorthand is possible (but not shown): combining the attribute declaration and the first equation into a single clause by inserting the equation right-hand side before the semicolon in the declaration.

JastAdd supports inter-type declarations [KHH+01] where attributes can be added to an existing class in a modular fashion. The target class for each attribute and equation is specified by qualifying its name with the target class name, e.g., `Expr` and `FieldName` above. The attribute is then woven into the class hierarchy generated from the abstract grammar.

An *inherited* attribute propagates the context downwards the AST. Consider the task to determine the enclosing `TypeDecl` for a `BodyDecl` node. A type declaration can tell all its enclosed `BodyDecls` that it is the enclosing `TypeDecl` declaration. This can be implemented through an inherited attribute using the following syntax:

```
inh TypeDecl BodyDecl.enclosingTypeDecl();
eq TypeDecl.getBodyDecl().enclosingTypeDecl() = this;
```

Equations for inherited attributes are broadcast to an entire subtree to eliminate trivial copy-rules. This subtree is explicitly selected using a child accessor (`getBodyDecl()` in this case). The equation should thus be read as: *define the value for the enclosingTypeDecl attribute in the entire subtree whose root is the node returned by getBodyDecl() in a block node. The value should be* this, *i.e., a reference to the type declaration node defining the equation.*

3.3 Circular attributes

Circular Reference Attributed Grammars [MH03] allow iterative fixed-point computations to be expressed directly using recursive equations. Cyclic dependences are allowed as long as there is a fixed-point that can be computed with a finite number of iterations. This is for example guaranteed if the values for each attribute on a cycle can be organized in a lattice of finite height and all semantic functions involved in the computation of these attributes are monotonic on the respective lattice.

Consider the code snippet below that detects circular class hierarchies. A `ClassDecl` optionally extends another class through explicit naming. The declaration of that superclass is reached through the type of the extends clause. We define a boolean attribute `isCircular()` which is circular if for instance

a class A **extends** B and B **extends** A. The values in this computation form a trivial lattice where **true** is *bottom* and **false** is *top*. The semantic function (equation) is monotonic since the result stays at bottom, unless a class is reached that does not extend another class in which case the value is raised to top.

```
ast ClassDecl ::= <Name:String> [Extends:TypeName];
syn boolean ClassDecl.isCircular() circular [true] =
  hasExtends() ? getExtends().type().isCircular() : false;
```

3.4 Refine attributes and methods

A common way to extend functionality in a modular fashion is to define new node types for language extensions and provide new equations that override existing behavior in a superclass. While this feature is extensively used in our compiler there is sometimes the need to refine an equation or method definition in a module. This is similar to overriding but the new behavior affects the same class instead of a subclass. Consider the code snipped below that refines the equation for FieldName.isFieldName(String name) in module Base. This is similar to *around advice* in AspectJ [KHH+01]. The new definition invokes the base implementation using Base.FieldName.isFieldName(name).

```
refine Base eq FieldName.isFieldName(String name) =
  name.equals("?") ?
    true : Base.FieldName.isFieldName(name);
```

4 JavaDemoTypes Base language

The focus of this paper is to present a technique to implement and extend the type system for Java 1.4. However, we believe that many of the ideas transfer directly to any statically typed object-oriented language. The technique is demonstrated by defining a tiny language, JavaDemoTypes, that captures the essence of the Java 1.4 type system while using only a few type checking related constructs. We have, however, used the same technique to successfully implement the extion for full Java 1.4 cover full Java 1.4.The same techniques have been used successfully to cover full Java 1.4 and there is a publically available implementation. JavaDemoTypes is presented as a small base language with reference and primitive types only. The language is then extended with new kinds of types such as array types and non-null types in a modular fashion. The implementation for the base language with modular extensions is publically available as well.

The main design principle behind the static semantic analyser is to use the AST as the only data structure and describe all computations as declarative attributes. This is the key to implementing modular extensions to static semantic analysis such as type checking, name analysis, definite assignment, and reachability analysis. We represent a type by its declaration which can then be used to lookup members in name analysis, and to compute the subtype relation during type checking. There is thus no need for separate symbol tables. We present modular name analysis and how to resolve contextually ambiguous names for Java 1.4 in another paper [EH06].

The JavaDemoTypes language allows type declarations, primitive types, null values, object instantiation, member fields, and access to the current object through *this*. JavaDemoTypes allows us to write a very scaled-down version of Java that demonstrates our techniques while still keeping the language small enough to be presented in the paper. We first describe the abstract grammar of the language and the parts of its API that are used during type checking. The base language is then extended with new kinds of types: first *array types* and then *non-null types*. We extend it later in the paper, as needed, to capture more language concepts used by the array type and non-null type extensions.

4.1 Language structure

The base grammar in Figure 4 has a few different kinds of type declarations. Besides user defined class declarations there are primitive types, the null type, and we also introduce a special *unknown* type that serves as the top type and is the supertype of all types. Each type declaration inherits a single type declaration and provides a list of body declarations. A type declaration may thus only define a single direct supertype but we describe an extension based on multiple supertypes in Section 6.

The base grammar allows member fields as the only kind of body declaration but has an implicit constructor with no arguments in the same way as Java 1.4. Later examples include explicit constructors and method declarations as well. The fields are initialized where declared and the assignment is checked using the subtype relation to ensure that the assignment is type safe. A class instance expression is used to create an object and there is an explicit null literal. Basic support for names include field names that can be qualified by this or another field. Primitive integer literals are supported as well.

We represent the type of an expression by a reference attribute from each expression node to a corresponding type declaration node. These references are then used to express other properties, e.g., a qualified name lookup may use such a reference to delegate the lookup of member fields to the type declaration matching the qualifier's type. All types used in a program must thus have a corresponding type declaration node in the AST.

```
ast Program ::= TypeDecl*;

ast abstract TypeDecl ::= <Name:String> Extends:TypeName
                    BodyDecl* DynamicTypeDecl:TypeDecl*;
ast ClassDecl : TypeDecl;
ast PrimitiveDecl : TypeDecl;
ast NullDecl : TypeDecl;
ast UnknownDecl : TypeDecl;

ast BodyDecl ::= <Modifiers:String>;
ast FieldDecl : BodyDecl ::= TypeName <Name:String> Expr;

ast abstract Expr;
ast QualName : Expr ::= Left:Expr Right:Expr;
ast TypeName : Expr ::= <Name:String>;
ast FieldName : Expr ::= <Name:String>;

ast This : Expr;

ast ClassInstanceExpr : Expr ::= TypeName;
ast NullLiteral : Expr ::=;
ast IntLiteral : Expr ::= <Value:int>;
```

Figure 4. Abstract grammar for the JavaDemoTypes base language.

4.2 Name binding and type system API

For brevity we only include the implementation of the base system that is concerned with types. The full implementation including name analysis, is available at [EH]. This section presents the API to base code that is used but not included in the paper. Figure 5 shows the relevant parts for this paper. The first two attributes are concerned with the type of an expression. Each expression has a type and that type is represented by a reference to its corresponding type declaration ①. There is a singleton type declaration used to indicate that there is a type error ②. That declaration is used as a null-object to automatically handle propagation of type errors. The next four equations deal with names. Each FieldName is bound to a FieldDecl ③. Some names are qualified by an expression ④. If this is the case, then there is a reference to that expression ⑤. There is also a reference from each Expr to the TypeDecl it is enclosed by ⑥. A type declaration can be queried for all member fields with a certain name ⑦ which is used for qualified lookup of names.

```
  // each Expr is bound to a type declaration
① syn TypeDecl Expr.type();
  // there is a singleton null-object named
  // unknown that indicates a type error
② inh TypeDecl Expr.unknown()

  // a FieldName is bound to a declaration
③ syn FieldDecl FieldName.decl()
  // expressions can be qualified ...
④ inh boolean Expr.isQualified();
  // ... and then have a qualifier
⑤ inh Expr Expr.qualifier();
  // all Expr are enclosed in a type declaration
⑥ inh TypeDecl Expr.enclosingType();
 // a type may have a member field named n
⑦ syn FieldDecl TypeDecl.memberField(String n);
```

Figure 5. The base language API that is used by the type extension.

4.3 Subtype computation for error checking

.

The *subtype relation* is used when typechecking object-oriented programs. Subsumption allows a subtype S of T to be used at all places where a type T is expected. A typical example is assignment where an expression is assigned to a variable and the type of that expression must be a subtype of the declared variable type. The JavaDemoTypes implementation performs error checking by collecting errors during a generic traversal of the AST and then presenting possible errors to the user. Figure 6 shows sample error checking for `FieldDecl` where the initialization is type checked ① and for `FieldName` where the name binding is checked ②. In the full Java compiler the error method adds location information to the error message, e.g., filename and row number.

Extensible subtype tests The typechecker uses an attribute `subtype` that checks whether two types are in the subtype relation.

```
syn boolean TypeDecl.subtype(TypeDecl type);
```

We use a straight-forward implementation of the subtype test for $S <: T$ that searches the direct supertypes of S transitively for T to determine if S and T are in a subtype relation. This works well in practice due to the automatic caching of previous subtype tests.

```
  public void FieldDecl.errorCheck() {
①   if(!getExpr().type().subtype(getTypeName().type()))
      error("Field␣" + getName() + "␣assignment␣error");
  }
  public void FieldName.errorCheck() {
②   if(decl() == null)
      error("Undefined␣field␣" + getName());
  }
```

Figure 6. Typical type checking and name binding error detection.

It is not always necessary to do this search since some combinations of types can not be in a subtype relation, e.g., a `ClassDecl` type can not be a subtype of a `PrimitiveDecl` type. It would also be somewhat inefficient to make the `NullType` have a set of direct supertypes that include all leaf `ClassDecls` in the system. We thus need to provide different equations for the subtype attribute depending on the kinds of types that are involved in the computation. Since both the receiver and the argument affect the selection of equation we need to dispatch not only on the run-time type of the receiver, but also on the the run-time type of the argument. We solve this problem by simulating a multi-dispatch mechanism to allow for modular specification of equations and also to allow extensibility when introducing new kinds of types.

We use the classical solution of two message dispatches, one to resolve the polymorphism of each involved type [Ing86]. Figure 7 shows the implementation of the subtype relation for the base language using double dispatch. The first invocation, subtype ①, reduces the polymorphic receiver to a monomorphic one by the type dispatch inherent in method invocations. The target method, e.g., ②, reduces the polymorphic argument into a monomorphic one by a second dispatch on that argument. Notice that this second dispatch reverses the relation from *subtype* to *supertype* while also selecting a specific supertype computation based on the run-time receiver type, e.g., ③ or ④.

It is also worth noticing that while the traditional use of double dispatch lacks modularity since multiple classes are affected, the use of inter-type declarations allows the double dispatch implementation to be modularized. We can thus provide a specific equation for an arbitrary combination of type kinds in the subtype relation.

4.4 Adding array types

Before we describe the non-null extension we will demonstrate how the base language can be extended with another kind of type: *array types*. This demon-

```
  // double dispatch pattern to implement binary methods
① syn boolean TypeDecl.subtype(TypeDecl type);
  eq ClassDecl.subtype(TypeDecl type)
    = type.supertypeClassDecl(this);
② eq NullDecl.subtype(TypeDecl type)
    = type.supertypeNullDecl(this);
  eq PrimitiveDecl.subtype(TypeDecl type)
    = type.supertypePrimitiveDecl(this);
  eq UnknownDecl.subtype(TypeDecl type)
    = type.supertypeUnknownDecl(this);

  // the subtype relation is reflexive and transitive
  syn boolean TypeDecl.supertypeClassDecl(ClassDecl type)
    = this == type || type.superclass().subtype(this);

  // all types are supertypes of NullDecl ...
③ syn boolean TypeDecl.supertypeNullDecl(NullDecl type)
    = true;
  // ... except for PrimitiveDecls
④ eq PrimitiveDecl.supertypeNullDecl(NullDecl type)
    = false;

  // all types are subtypes of UnknownDecl (top)
  syn boolean TypeDecl.supertypeUnknownDecl(UnknownDecl t)
    = this == t;
  eq UnknownDecl.supertypeClassDecl(ClassDecl t) = true;
  eq UnknownDecl.supertypeNullDecl(NullDecl t) = true;
  eq UnknownDecl.supertypePrimitiveDecl(PrimitiveDecl t)
    = true;

  syn boolean TypeDecl.supertypePrimitiveDecl(
    PrimitiveDecl type) = this == type;
```

Figure 7. Base language subtype relation computation.

strates a general technique to extend the typechecker in a modular fashion and will be used for the non-null types as well. A Java array is an object that contains a number of variables that do not have names but are selected using an index instead. These variables are called the components of the array and have the same static type which is called the *component type* of the array. If the component type of an array is `T`, then the type of the array itself is written `T[]`. The component type of an array may itself be an array type and thus provide arrays

of arbitrary dimension. The type declaration that is reached when there are no more component types of an array type is called the *element type* of the array type.

The base language is extended with arrays through the following extensions:

- An array creation expression
- An expression to access components in an array
- Type names to represent array types
- A type declaration representing array types
- Extend the subtype relation to handle array types

The first three extensions correspond to new syntactic constructs, while the last two concern extensions to the type checking and name analysis modules in the compiler. There are also a few attributes that need to be given equations for the new syntactic contructs to integrate with existing computations, e.g., to define the type of the new expressions.

Array type representation The abstract grammar for the base language is extended with the following array specific node types:

```
ArrayCreationExpr : Expr ::= TypeName Expr*;
ArrayAccess : Expr ::= Expr Index:Expr;
ArrayTypeName : TypeName ::= TypeName;
ArrayDecl : ClassDecl;
```

There is an important difference between `ArrayDecls` and other compiler added type declarations. The maximum dimension of an array is not a static property but varies from program to program. New `ArrayDecls` are therefore added lazily when a particular dimension is requested. Figure 8 provides two operations that relate a component type to its array type. The `arrayType()` attribute ② of a type declaration returns the array declaration with a component type that is the type of the attribute receiver. The `componentType()` attribute ① decreases the dimension of the receiving array type declaration by one. Since array type declarations in a sense are a function of another type declaration we add the lazily created `ArrayDecl` as a child to the `DynamicTypeDecl` list in the type declaration for its component type ③.

These operations are then used to define the type attribute for the new expressions as shown in Figure 9. An `ArrayTypeName` is a `TypeName` followed by a pair of brackets. The type of an `ArrayTypeName` is the array type of its component type ①. Since an `ArrayTypeName` is itself a `TypeName`, this language construct allows for `ArrayTypeNames` of arbitrary dimension. The type of an `ArrayAccess` is the component type of the array object that is accessed

```
① inh TypeDecl TypeDecl.componentType();
  // the component type of an array is the type of the
  // declaration it is based on
  eq TypeDecl.getDynamicTypeDecl().componentType() = this;
  // the component type of a class decl is the unknown type
  eq Program.getTypeDecl().componentType() = unknown();

② syn lazy TypeDecl TypeDecl.arrayType() {
    // create an array decl with a
    // member field int length
    TypeDecl typeDecl = new ArrayDecl( ... );
③   addDynamicTypeDecl(typeDecl);
    return typeDecl;
  }
```

Figure 8. Operations to increase and decrease the arity of an array type.

②. An `ArrayCreationExpr` specifies an element type and then a list of expressions in brackets that define the size and dimension of the array. Each pair of brackets increases the dimension by one ③.

```
  // bind to an array type matching this name
  // each bracket pair increases the dimension by one
① eq ArrayTypeName.type() = getTypeName().type().arrayType();

  // the type is the type of the array component
② eq ArrayAccess.type() = getExpr().type().componentType();

  // a name followed by a list of bracketed int values
  // each bracket pair increases the dimension of the array
  eq ArrayCreationExpr.type() {
    TypeDecl typeDecl = getTypeName().type();
③   for(int i = 0; i < getNumExpr(); i++)
      typeDecl = typeDecl.arrayType();
    return typeDecl;
  }
```

Figure 9. Type equations for array related expressions.

The final step is to extend the subtype relation to handle array types with new equations as shown in Figure 10.

```
// double dispatch
eq ArrayDecl.subtype(TypeDecl type)
  = type.supertypeArrayDecl(this);
syn boolean TypeDecl.supertypeArrayDecl(ArrayDecl type)
  = false;

// SC[] is a subtype of TC[] if SC is a subtype of TC
eq ArrayDecl.supertypeArrayDecl(ArrayDecl type) =
  this == type
  || type.componentType().subtype(componentType());

// array types are subtypes of UnknownDecl (top)
eq UnknownDecl.supertypeArrayDecl(ArrayDecl type)
  = true;
```

Figure 10. Modular extension of the subtype relation to include array types.

5 Non-null types extension

This section presents a modular non-null type extension to Java based on the work by Fähndrich and Leino [FL03] introduced in Section 2. The main purpose is to detect dereferenced null pointers but also to allow the developer to better show his or her intent explicitly through annotations in the code.

The base language is extended with non-null types through the following extensions:

- A type declaration representing non-null types
- Extend the subtype relation to handle non-null types
- Annotate a declaration with the non-null property
- Refine type binding rules
- Detect possible null pointer violations

A new kind of type declaration, non-null type declaration, is added to describe a type that represents class declarations but not the null type. The subtype relation computation is extended to cope with this new kind of type declaration. Non-null types are specified in the source code by annotating field declarations

with a modifier [NotNull]. The type binding rules need to be refined to take these modifiers into account and also to bind the type of newly instantiated objects to a non-null type declaration since they are guaranteed to be non-null. Finally the error checker needs to be extended to detect possible null pointer violations. This is done by adding a check with an appropriate error message for a possibly-null type at each expression where the language stipulates the VM to throw a null pointer exception if the expression evaluates to null.

5.1 Non-null type representation

Each type in the system is required to have a corresponding type declaration. Classes are explicitly declared and can thus be used as is to represent types. A non-null type is not explicitly declared but need to be added by the compiler. We new show how to dynamically build a suitable structure to represent the non-null counterpart of each user-defined class declarations. A non-null counterpart to each user-defined class declaration is built in a way very similar to the array type declaration in Section 4.4. Figure 11 shows how to find a non-null counterpart of a possibly-null declaration ② and vice versa ①. Since a field declared to be of an array type may have a non-null modifier as well there is interaction between array types and non-null types. We choose to always keep array types closest to the element type and then add the non-null declaration node below all array specifiers. To keep this invariant the `arrayType()` and `componentType()` attributes are overridden for `NonNullDecl`. When creating an `arrayType()` of a `NonNullDecl` we first take its possibly-null counterpart and then increase its dimension and finally add the non null property again to group all array declarations closest to the element type ③. Array types only have a single member, the length field, besides the members inherited from Object. `NonNullTypes`, on the other hand, have the same members as its possibly-null counterpart. We therefore delegate all member lookups to the possibly-null type ④.

5.2 Extend subtype relation to non-null types

The subtype relation needs to be extended to handle the new non-null types as shown in Figure 12. The equation for `supertypeNullDecl(NullDecl type)` ① is overridden for `NonNullDecl` to be false, effectively removing the `NullDecl` and `NonNullDecl` pair from the subtype relation.

5.3 Refine type binding and error checking

With the subtype relation in place and attributes to find the non-null counterpart of a possibly-null type and vice versa, the final step is to refine a few

```
  // link possibly-null type and the non-null counterpart
① inh TypeDecl TypeDecl.possiblyNull();
  eq TypeDecl.getDynamicTypeDecl().possiblyNull() = this;
  eq Program.getTypeDecl().possiblyNull() = unknown();

② syn lazy TypeDecl TypeDecl.nonNull() {
    TypeDecl typeDecl = new NonNullDecl(getName() + "-",
      new TypeName(getExtends().getName()), new List(),
      new List()
    );
    addDynamicTypeDecl(typeDecl);
    return typeDecl;
  }

  // preserve invariant that ArrayDecl nodes are closer
  // to the element type declaration than the NonNullDecl
③ eq NonNullDecl.arrayType()
    = possiblyNull().arrayType().nonNull();
  syn TypeDecl NonNullDecl.componentType()
    = possiblyNull().componentType().nonNull();

  // delegate member field lookup to possibly-null type
④ eq NonNullDecl.memberField(String name)
    = possiblyNull().memberField(name);
```

Figure 11. Compute the non-null counterpart of a possibly-null type and vice versa.

type equations and to add checks for possible null-pointer volations. Figure 13 shows the implementation of refined type equations for both `FieldDecl` and `Class-InstanceExpr`. The type of a field declaration is non null if there is an explicit `[NotNull]` modifier ①. The base type is reached through the `TypeName` and rebound to its non-null counterpart using the `nonNull()` attribute. All newly instantiated objects are guaranteed to be non null and therefore bound to a non-null type in a similar manner ②. These new types are then used to check for possible dereferenced null pointers in field names. If the qualifier to a field name is of a possible-null type there might be a null pointer violation at run-time and we instead raise a compile-time error ③. The extended subtype attribute ensures that a field declared non-null can not be assigned a possibly-null typed value.

```
  // subtype rules
  eq NonNullDecl.subtype(TypeDecl type)
    = type.supertypeNonNullDecl(this);
  syn boolean TypeDecl.supertypeNonNullDecl(NonNullDecl t)
    = false;

  // a type S- is a subtype of a type T
  // if its counterpart S is a subtype of T
  eq ClassDecl.supertypeNonNullDecl(NonNullDecl type)
    = type.possiblyNull().subtype(this);
  // a type S- is a subtype of a type T-
  // if type S is a subtype of type T
  eq NonNullDecl.supertypeNonNullDecl(NonNullDecl type)
    = type.possiblyNull().subtype(possiblyNull());
  // a possibly-null type is not a subtype of a non-null
  eq NonNullDecl.supertypeClassDecl(ClassDecl type)
    = false;
  // non-null types are subtypes to the unknown type
  eq UnknownDecl.supertypeNonNullDecl(NonNullDecl type)
       = true;
  // null is not a subtype to a non-null type
① eq NonNullDecl.supertypeNullDecl(NullDecl type) = false;
```

Figure 12. Modular extension of the subtype relation to include non-null types.

5.4 Raw types

The extensions done so far assume that all fields have been assigned non-null values before they are accessed. This may seem like a safe assumption since the fields in the language described so far require immediate initialization and there is a subtype check that verifies that the assigned value is non null. However, constructors in combination with virtual methods complicates the problem as illustrated by the example in Figure 3 discussed in Section 2. Figure 14 shows the abstract grammar for the constructor and method declarations as well as expressions and statements required to implement a similar example in JavaDemoTypes.

The problem in the example is that `this` references a partially initialized object within a constructor. We introduce a raw type to reference partially initialized object for which we can not assume that any fields are non-null (or even initialized). A raw object originates in a constructor as `this` but can escape the constructor when being the receiver of a method invocation, an argument in a method invocation, or the right hand side in an assignment. We allow methods

```
  // detect [NotNull] modifier
① refine Base eq FieldDecl.type()
    = modifier("[NotNull]") ?
    getTypeName().type().nonNull() : getTypeName().type();
  syn boolean BodyDecl.modifier(String s)
    = getModifiers().indexOf(s) != -1;

  // instantiated objects are non-null
② refine Base eq ClassInstanceExpr.type()
    = Base.ClassInstanceExpr.type().nonNull();

  // detect attemt to dereference possibly-null types
  refine BaseErrorCheck void FieldName.errorCheck() {
    BaseErrorCheck.FieldName.errorCheck();
③   if(isQualified() && qualifier().type().mayBeNull())
      error("Qualifier␣may␣be␣null");
  }
  syn boolean TypeDecl.mayBeNull() = true;
  eq NonNullDecl.mayBeNull() = false;
```

Figure 13. Refine type binding and error checking when using non-null types.

```
ConstrDecl : BodyDecl ::= <Name:String> Block;
MethodDecl : BodyDecl ::= TypeName <Name:String>
                          ParamDecl* Block;
ParamDecl ::= <Modifiers:String> TypeName <Name:String>;

ParamName : Expr ::= <Name:String>;
MethodInv : Expr ::= <Name:String> Arg:Expr*;
AssignExpr : Expr ::= LValue:Expr Expr;

abstract Stmt;
Block : Stmt ::= Stmt*;
ExprStmt : Stmt ::= Expr;
```

Figure 14. Extend the base language with constructors, methods, and a few statements.

to be declared raw which means that the type of `this` is raw in that context which in turn means that fields read in the current object should be considered possibly null. We introduce a new type declaration, `RawDecl : TypeDecl` and extend the subtype relation for raw types (using the same implementation

technique as for non-null types and array-types) with the following rules:

$$A <: A^{raw}$$

$$B^{raw} <: A^{raw} \quad iff \quad B <: A$$

Both possibly-null and non-null types are thus subtypes of the corresponding raw type. This also means that raw types are not subtypes of $Object$ but rather the $Object^{raw}$ type.

In the example in Figure 3 discussed in Section 2 the `print()` method would have to be declared `[Raw]` or else the call from the constructor in `A` would result in a type error since `Araw` is not a subtype of `A`. If we had dereferenced `s` in that context there would be a possible null pointer violation and we would have to guard that operation with an explicit not-null comparison `s != null`.

Detect partially initialized objects The type of the implicit and explicit **`this`** expression differs depending on its context, e.g., in a constructor body or explicitly declared raw method body. Figure 15 shows the implementation of the refined type of **`this`** ① for various contexts. The `thisType()` attribute ② is defined to reference the type declaration of **`this`** in a particular context. Equation ③ sets the type to be raw in the constructor body. Equation ④ delegates to equation ⑥ which defines the raw property for a method declaration. The Java language specification prescribes that fields in the same class can not be used in an initialization of another field unless they are declared before that field. This actually allows us to to consider the type of **`this`** in the initialization of a field not to be raw ⑤.

```
① refine Base eq This.type() = thisType();

② inh TypeDecl This.thisType();
③ eq ConstrDecl.getBlock().thisType()
      = enclosingType().raw();
④ eq MethodDecl.getBlock().thisType() = methodThisType();
⑤ eq FieldDecl.getExpr().thisType() = enclosingType();
  eq Program.getTypeDecl().thisType() = null;

⑥ syn TypeDecl MethodDecl.methodThisType()
     = modifier("[Raw]") ?
       enclosingType().raw() : enclosingType().nonNull();
```

Figure 15. Compute the rawness state of *this* in a certain context.

A raw-typed reference may only be used as the receiver of a method if it is a subtype of the `thisType()` of that method. This constraint is implemented by a check ③ in Figure 16. The type of a field name needs to be refined as well to take rawness into account. If the qualifier for a field name is raw then the field is always possibly null regardless of the declared type of that field. However, this rule only applies to field names that do not act as *lvalues* (the left hand side) in an assignment since we never assign a possibly-null value to a variable declared non-null. The type of a field name is refined by equation ① in Figure 16 and the lvalue detection by equation ②. This ensures that as soon as a non-null field is assigned it can never reference null. Since the initalization of a field is actually done within a constructor we do not have to require immediate initialization of non-null fields but merely require them to be definitely assigned at the end of every constructor. This kind of computation needs to be performed by a Java compiler anyway and can thus be reused in this new context.

5.5 Casts using explicit null comparisons

Explicit casts are useful to allow the developer to provide hints to a type system that he or she considers too restrictive. This is usually combined with a dynamic check to ensure safe execution. When introducing non-null types it would certainly be useful to convert a possibly-null value into its non-null counterpart. Instead of introducing new syntax to the language, we use an explicit comparison with null to change the type of a parameter or local variable in a particular context. Consider the non-null example in Figure 1. The comparison ensures that `n` is non-null at the beginning of the conditional block. We can then safely assume that `n` is non-null in this context, as long as `n` is not assigned a possibly-null value. This is only vaild for parameters and local variables since they can only be changed locally in the current method. The guard is only valid as long as the variable is not assigned possibly-null values in the guarded branch.

Figure 17 defines an inherited attribute `guardedByNullCheck(String n)` ② that is true when there is an explicit comparison to not null and no possibly-null assignments in the executed branch ③. That attribute is then used to refine the type of `ParamName` taking its context into account ①. The attribute delegates to the nested environment to allow nested guards for different variables ④. To check for possibly-null assignments we do a generic traversal that does not take control flow into account which may be unnecessarily convervative but which is simple to implement ⑤.

6 Discussion

The same technique as described in the previous sections have been used to implement a full non-null type extension to Java 1.4. The complete implemen-

```
  refine Base eq FieldName.type() {
    TypeDecl qualifierType = isQualified() ?
      qualifier().type() : thisType();
    TypeDecl type = Base.FieldName.type();
①   if(!type.mayBeNull()
         && (qualifierType instanceof RawDecl)
         && !isLValue()) {
      type = type.possiblyNull();
    }
    return type;
  }

② inh boolean FieldName.isLValue();
  eq AssignExpr.getLValue().isLValue() = true;
  eq AssignExpr.getExpr().isLValue() = false;
  eq QualName.getLeft().isLValue() = false;
  eq Program.getTypeDecl().isLValue() = false;

  refine BaseErrorCheck void MethodInv.errorCheck() {
    BaseErrorCheck.MethodInv.errorCheck();
    if(decl() != null) {
      TypeDecl qualifierType = isQualified() ?
        qualifier().type() : thisType();
      if(!qualifierType.subtype(decl().methodThisType()))
③       error("Qualifier␣not␣compatible");
    }
  }
```

Figure 16. Refine type rules to take rawness into account.

tation is roughly 220 lines of code. The main differences are that we need to extend error checking, e.g., overriding of methods with the new rules, and to deal with interfaces. Interfaces complicates the subtype relation in that a class may have several direct supertypes. The implementation needs to be changed to check not only a single supertype but a set of supertypes. We add interfaces as a new kind of node type and provide equations for that node type similar to the equations in Figure 7 in Section 4.3.

The combination of declarative attributes and using the AST as the only data structure has proven useful from a modularity point of view for both name analysis and type checking purposes. We have shown how to extend the subtype relation in a modular fashion, dynamically create new types that are not explicitly declared and refine existing compuations to use these new kinds of

```
  refine Base eq ParamName.type() {
    TypeDecl type = Base.ParamName.type();
①   if(type.mayBeNull()&&guardedByNullCheck(getName())){
      type = type.nonNull();
    }
    return type;
  }
② inh boolean ParamName.guardedByNullCheck(String name);
  inh boolean IfStmt.guardedByNullCheck(String name);
  eq IfStmt.guardedByNullCheck(String name) {
③   if(getCond().isVarNotEqualNull(name)
          && !getThen().assignsVarToPossiblyNull())
      return true;
④   return guardedByNullCheck(name);
  }
  eq MethodDecl.getBlock().guardedByNullCheck(String n)
    = false;
  eq ConstrDecl.getBlock().guardedByNullCheck(String n)
    = false;

  // traverse the subtree and detect null assignments
⑤ boolean ASTNode.assignsVarToPossiblyNull(ParamDecl p){
    for(int i = 0; i < getNumChild(); i++)
      if(getChild(i).assignsVarToPossiblyNull(p))
        return true;
    return false;
  }
  boolean AssignExpr.assignsVarToPossiblyNull(ParamDecl p) {
    if(getLValue().decl() == p && !getExpr().isNonNull())
      return true;
    return super.assignsVarToPossiblyNull(p);
  }

  // true when the expression is a comparison that
  // a variable named name does not equal null
  syn boolean Expr.isVarNotEqualNull(String name);
```

Figure 17. Cast a possibly-null typed variable into a non-null typed variable by explicit comparison to not-null.

types.

We believe the same techniques can be used with minor modifications to implement other type based extensions. Javari [TE05] is a type system that

is capable of expressing and enforcing immutability constraints. The state of the object to which an immutable reference refers cannot be modified using that reference. This property is enforced transitively, i.e., an object referenced through an immutable reference only exposes immutable references to other objects. The subtype relation and an immutable counterpart to each mutable type can be built similar to the non-null example. However, the dynamically built immutable type declaration should not delegate lookups to the base type declaration directly, but should rather create new method and field signatures with immutable type declarations. This will implement the transitive property of Javari described above. We are currently implementing Java 5 generics including wildcards using the same technique. Generic types are instantiated with type arguments and we build method and type signatures that reflect the instantiated type parameters.

7 Automatic non-null type refinement

This section presents a technique to refine types in library legacy code, e.g., the JDK, to include non-null annotations. The inclusion of legacy code in the approach is important for several reasons. First, it allows the user to start using non-null types on new code, while reusing existing non-annotated libraries, and at the same time not having to guard all uses of the legacy code with null-checks. Second, it allows the user to gradually refactor legacy code to explicitly annotated code, moving the barrier between annotated and non-annotated code. The approach can also be used to detect possible null pointer related errors in the legacy code: If a reference that is inferred to be possibly null is dereferenced, this might be the source of runtime errors.

We first describe how types are automatically refined to non-null types without violating the extended subtype rules, not taking partially initialized objects into account. We then extend the implementation to infer raw types and finally add support for casts using explicit null comparisons.

7.1 Infer the non-null type property

The purpose of the non-null type refinement is to infer the non-null property for declarations instead of the explicit annotations used in Section 5. The goal is to turn as many possibly-null references into non-null references as possible while not violating the extended type system.

The implementation in Figure 18 provides an inferred property that corresponds to each annotation in the non-null extension. There is a direct correspondence between the implementation of that property and the extended type checking rules in the extended type system. Each `ParamDecl` has an attribute

`isNonNull()` ① that infers the non-nullness property instead of explicit declaration through modifiers. This property is only valid when all its assigned values are non null ②. This corresponds to the new subtype rules in Figure 12. The `FieldDecl` is somewhat more complicated since it has additional type checking rules besides the new subtype rules. A `FieldDecl` can thus only be non-null if it is guaranteed to be definitely assigned after each constructor ③ besides the limits caused by subtype rules.

We do allow violations of the refined error check that detects dereferenced possibly-null references. If we could statically prove the absence of this kind of error using non-null type refinement, there would be no need for annotations. Since this is not the case, we accept possible null-pointer exceptions in legacy code. The main purpose of automatic type refinement is to allow new properly annotated code to use legacy code without the need for type casts through comparisons with null.

Both `ParamDecl` and `FieldDecl` depend on the non-null property of the expression values that they are assigned to. That property needs therefore to be computed for expressions as well ④. Newly created objects are non-null and use a direct equation ⑤ similar to Figure 13. `FieldNames` need the rawness property to be computed to allow for safe refinement to non-null types ⑥. The rawness property computation is described in Figure 19. `ParamNames` that are not refined to non-null may still be considered non-null when guarded by an explicit null comparison ⑦. This property is described in Figure 7.3.

The `isNonNull()` computation often leads to circular dependences. The JastAdd evaluation engine has direct support for circular attributes through iteration until a fixed point is reached. Consider the following code snippet for an element in a linked list:

```
class Element {
  Element next;
  Element pred;
  void remove() {
    pred.next = next;
  }
}
```

The assignment of the field *next* depends directly on itself and the nullness computation is thus circular. The boolean nullness property can be represented as a trivial lattice of height two. The finite height of the lattice guarantees that the compuation will terminate and reach a fix-point. The `isNonNull()` attribute is declared to be circular and its *bottom* value is set to **`true`**. A single possibly null value will raise the attribute value to **`false`** which is *top*.

```
① syn boolean ParamDecl.isNonNull() circular [true] {
    for(Iterator iter=assignedValues(); iter.hasNext();){
      Expr expr = (Expr)iter.next();
②     if(!expr.isNonNull())
        return false;
    }
    return true;
  }

  syn boolean FieldDecl.isNonNull() circular [true] {
    for(Iterator iter=assignedValues(); iter.hasNext();){
      Expr expr = (Expr)iter.next();
      if(!expr.isNonNull())
        return false;
    }
    for(Iterator iter = enclosingType().constructors();
        iter.hasNext(); ) {
      ConstrDecl constr = (ConstrDecl)iter.next();
③     if(!constr.definitelyAssigns(this))
        return false;
    }
    return true;
  }

④ syn boolean Expr.isNonNull() circular [true];
⑤ eq ClassInstanceExpr.isNonNull() = true;
⑥ eq FieldName.isNonNull() = decl().isNonNull()
      && !qualifier().isRaw();
⑦ eq ParamName.isNonNull() = decl().isNonNull()
      || guardedByNullCheck(getName());

  // assigned in all paths through the constructor
  syn boolean ConstrDecl.definitelyAssigns(FieldDecl f);
  // references to all assigned expression values
  syn Set FieldDecl.assignedValues();
  syn Set ParamDecl.assignedValues();
```

Figure 18. Infer the non-null property.

7.2 Infer raw types for partially initialized objects

Raw types need to be computed to make the use of fields refined to be non-null safe. Instead of annotating a method as raw when it is accessed from a con-

structor or a raw context we infer that property automatically. A `MethodDecl` is raw if there is at least one receiver that it is invoked from that is raw. Figure 19 shows the implementation of this property which is circular, similar to the `isNonNull()` property. This equation corresponds to the explicit subtype check extended with raw types. Since the receiver is an expression it also needs to have its raw property computed. The computation for `thisTypeRaw()` is very similar to the implementation in Figure 15 and differs in that a boolean property is computed rather than an actual type and also in the possibility of circularities.

```
  syn boolean MethodDecl.isRaw() circular [false] {
    for(Iterator iter=receiverExprs(); iter.hasNext();){
      Expr expr = (Expr)iter.next();
①     if(expr.isRaw())
        return true;
    }
    return false;
  }

② syn boolean Expr.isRaw() circular [false];
  eq This.isRaw() = thisTypeRaw();

③ inh boolean This.thisTypeRaw() circular [false];
  eq MethodDecl.getBlock().thisTypeRaw() = isRaw();
  eq ConstructorDecl.getBlock().thisTypeRaw() = true;

  // references to all expression values that
  // qualifies invocations of this method
  syn Set MethodDecl.receiverExprs();
```

Figure 19. Infer the raw property.

7.3 Infer type casts through explicit null comparisons

An explicit comparison with null is used as a cast in the new type system to convert a possibly-null type into its non-null counterpart. However, we believe that this syntax is quite natural and probably used in legacy code for the same purpose. Therefore, the property is computed using the same conservative requirements as in Section 5.5 when refining parameter types. The only difference

compared to Figure 17 is that the `guardedByNullCheck(String name)` attribute is circular.

8 Evaluation

We have presented JavaDemoTypes and shown how to extend its type system with non-null types. We have also shown a type refinement implementation that infers non-null types in legacy code. This section evaluates these techniques from a code size, execution time, and inference result point of view.

8.1 Code size

JavaDemoTypes was presented as a small kernel language, suitable to illustrate type system extensions. The same implementation technique is however used to implement the corresponding extension to the JastAdd Extensible Java compiler. We therefore compare the size of the implementations to illustrate how the techniques scale up to a full language.

Figure 20 compares the size of the modules in the base langauge for JavaDemoTypes to the corresponding implementation in the JastAdd extensible Java compiler. It is worth noticing that only a small subset of the full compiler is included in the comparison. The complete Java compiler is close to 14.000 lines of code.

Name	Demo	Java	Responsibility
TypeSystem	85	755	Compute subtype relation and expression types. Add run-time representation of primitive types.
NameBinding	65	1397	Simple name binding for types, fields, and methods including inheritance.
ErrorCheck	62	806	Detect undeclared names. Perform type checking of assignments, method invocations, etc.

Figure 20. The base compiler modules. The Demo column shows the size for JavaDemoTypes while the Java column shows the size for the corresponding functionality in the JastAdd Extensible Java compiler.

The implementation of the non-null language extension was presented in Section 5. Figure 21 compares that implementation to the full non-null extension for Java. The full implementation includes checks that ensures that modifiers are only used in valid contexts, and also include support for *raw-upto*

types. Even when this functionality is included, the full extension is only 56% larger than the extension for JavaDemoTypes. The extra effort to support the full language compared to the demo subset is relatively small compared to the huge difference in base system size.

Name	Demo	Java	Responsibility
NonNull	43	72	Add run-time representation of non-null types. Extend subtype relation. Include non-null modifiers in type computations. Detect unsafe dereferences.
Raw	73	99	Add run-time representation of raw types. Extend subtype relation. Include raw modifiers in type computations.
Guard	26	30	Check if a possibly-null variable is guarded by explicit comparison to null.
Check	n/a	20	Check for valid modifiers. Not included in JavaDemoTypes..
Total	142	221	

Figure 21. The non-null type extension. The Demo column shows the size for the modules presented for JavaDemoTypes while the Java column shows the size for the corresponding extension to the JastAdd Extensible Java compiler.

Section 7 presents a type refinement algorithm that infers non-null types in legacy code. That code is compared to the corresponding code for the full compiler in Figure 22. While still being fairly compact (< 3.3 % of the full compiler size), the refinement algorithm for full Java is roughly ten times larger than for JavaDemoTypes. The reason is that the type refinement extension is not as deeply integrated in the compiler as the non-null type extension, but merely adds additional computations on top of exisiting analyses. The base compiler converts and propagates types according to the operations in an expression tree. The type extension reuses that code for new types as well, e.g., null-null types and raw types. When we infer the non-null property, on the other hand, we need to implement that propagation in all language constructs in the base Java compiler. This is the main reason why the refinement algorithm is so much larger for the full compiler than for the demo subset. Some significant computations, e.g., the references from definitions to uses, were not included in the presentation but are included in the comparison for completeness.

Name	Demo	Java	Responsibility
NonNull	19	112	Infer the non-null property.
Raw	14	134	Infer the raw property.
Guard	8	45	Detect variables guarded by explicit null check.
Output	n/a	30	Include inferred properties in pretty printer.
Def-Use	n/a	113	Compute references from definitions to uses. This includes assignments, return values, method arguments, etc.
Flatten	n/a	24	Flatten the method hierarchy by conservatively merging methods with the same signature.
Total	41	458	

Figure 22. The type refinement extension. The Demo column shows the size for the modules presented for JavaDemoTypes. That code is for illustration purposes and not complete. The Java column shows the size for the full extension to the JastAdd Extensible Java compiler.

8.2 Inference performance

We have evaluated the type refinement algorithm using a substantial part of the JDK standard class library as a benchmark and measured the inferred non-null property in that body of code. Roughly 100.000 source lines from the following packages in JDK 1.4.2 were included in the analysis; *java.lang, java.util, java.io*.

Our primary use of the inference algorithm is to automatically detect non-nullness properties of the APIs in legacy code. When we annotate new code we would otherwise have to assume all legacy code to return possibly-null types. We are thus mostly interested in the non-nullness property for return values. The *% Non-null return* column in Figure 23 shows the percentage of reference typed return values that the inference algorithm concludes are non null. A substantial part of the return values can thus be considered non null when used by annotated code instead of assumed possibly null.

However, additional insights can be gained from the analysis. We can use the inferred types of expressions to find possible null-pointer violations in the legacy code. We thus partition the code into blocks that are safe from null-pointer violations and blocks where null pointers may occur. The *% Safe dereferences* column in Figure 23 shows the percentage of locations in the code where the language prescribes a possible null pointer exception, but where the inference algorithm inferred the non-nullness property. The *# Safe dereferences* column shows the actual number of dereferenced non-null references.

Raw types allow for safe non-null declaration of instance fields. Without this analysis all fields would have to be considered possibly-null. To see if

	Non-null return		Safe dereferendes	
	%	#	%	#
Non-null + raw	24 %	259	71 %	8580
Non-null	24 %	252	69 %	8354

Figure 23. The result from non-null inference in terms of sucessful type refinements

we benefit from this analysis we compute the same property with raw types disabled and consider all fields possibly-null. The first line shows the results with raw types enabled while the second line shows the result without raw types. Only a few fields are inferred non-null but they do affect the number of non-null return values, although this is only a minor effect. For this analysis to be safe we are required to also infer the raw property, to detect fields that can possibly be accessed prior to their initialization. Otherwise we assume that no non-null fields are accessed before they are initialized. This is not safe for partially initialized objects.

8.3 Refinement speed

To evaluate the speed of our inference algorithm we compare the execution times for our inference implementation, the base compiler, and the standard javac compiler for reference. Figure 24 shows the execution times for our benchmark as described above. We included the time for javac as a reference to show that the JastAdd design to structure the compiler is reasonably fast while still providing excellent support for modularity and extensibility. We conclude that the execution speed for the analysis is reasonably high even for fairly large programs.

	NonNull + Raw	Non Null	Base Compiler	javac
time	9.1 s	9.0 s	6.4 s	2.5 s

Figure 24. Execution speed with and without various type refinement computations

9 Related work

The focus of this paper is to show how the type system in a Java 1.4 compiler can be extended in a modular fashion using JastAdd. Another tool for extend-

ing Java is the Polyglot Extensible Compiler Framework [NCM03]. The base compiler has been extended with various type related concepts, e.g., parameterized types [BLM97] and predicate dispatch [Mil04]. This is a tool based on imperative programming and visitor patterns and relies on explicit passes to schedule computations. In contrast, we base our work on the JastAdd system which supports declarative extensions through attribute grammars. To our knowledge, there is no implementation of non-null types in Polyglot.

We base our type extension on the work on non-null types for object-oriented languages, as presented by Fähndrich and Leino [FL03]. Their approach differs from earlier work on non-null types by taking inheritance and object initialization into account and they did a prototype implementation for C#. We have implemented a modular extension of non-null types for Java and extend their approach by adding a simple but effective inference algorithm in order to handle legacy code that does not have non-null annotations.

Type qualifiers [FFA99] allow types to be refined by adding a qualifier to a type name, e.g., non-null. This is a simple but highly useful form of subtyping. A framework is presented that extends the type rules in C to propagate the qualifiers through a program. There are publically available systems with extensible type qualifiers without any automatic soundness proofs with implementations for C (CQUAL) and Java (JQUAL). Semantic Type Qualifiers [CMM05] provide an impressive framework and language to specify and automatically prove soundness of extended types by refining types using type qualifiers. An automatic therorem prover is used to prove soundness of various properties including non-null types for C. Our implementation is larger than the corresponding type qualifier definitions but we can take other properties than propagation of types into account when refining types, e.g., definite assignment, null comparisons, etc. We are not aware of any support for non-null types for object-oriented languages that take partially initialized objects into account in these systems.

10 Conclusions and Future work

We have shown how the JastAdd extensible Java compiler can be extended with non-null types in a modular fashion. The implementation as well as the core langugage used to demonstrate the techniques are publically available at the JastAdd website [EH]. A type refinement implementation that automatically infers non-null types in legacy code has also been presented. Our implementation constitutes a strong case for ReCRAGs and the JastAdd system, demonstrating how the declarative features of reference attributes and circular attributes can be taken advantage of to provide a compact modular implementation of a non-trivial type-system addition. The resulting implementation, including both the

type system extension and the automatic refinement, is only around 700 lines of JastAdd code, and is available at the JastAdd website.

Future work includes some refinements of the implementation. Currently, *raw upto* is used in the non-null type extension, but not yet in the automatic type refinement implementation. The implementation will also be extended to support elements in arrays as described by Fähndrich and Leino [FL03]. References to arrays are handled in the same way as other references but the components of an array are always possibly null in our implementation. Fähndrich and Leino use a dynamic cast to convert an array with possibly-null component references into its non-null counterpart. This requires a run-time check with comparison to null for each array component.

There are other approaches than type systems that can detect possible dereferenced null references. Points-to-analysis for Java computes the set of objects that a reference can point to [Ste96,BLQ+03,SGSB05]. That analysis can thus be used to detect whether a reference may be null or not. These computations usually take a lot more context into account than the type based approach with non-null types and thus require more resources. However, it would be interesting to combine points-to-analysis with our type refinement computation to refactor legacy code. There will certainly be references that are considered non-null by the points-to-analysis and possibly-null by the less precise type refinement algorithm. It would be easy to change the refinement algorithm to return the reason why a certain reference is not refined to non-null. The code could then be refactored manually to fulfil that condition and be declared non-null.

It would also be interesting to use the presented techniques to implement other type system extensions, e.g., reference immutability in Javari [TE05]. We are currently evaluating the approach on a more demanding type system extension by extending the base Java 1.4 compiler with Generics including wildcards from Java 5.

References

[BLM97] Joseph A. Bank, Barbara Liskov, and Andrew C. Myers. Paramterized types and java. In *POPL'97*, 1997.

[BLQ+03] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 103–114. ACM Press, 2003.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*. Springer, 2004.

[Bon] Daniel Bonniot. Why programs written in Nice have less bugs. http://nice.sourceforge.net/safety.html.

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

[CMM05] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. *SIGPLAN Notices*, 40(6), 2005.

[EH] Torbjörn Ekman and Görel Hedin. The JastAdd compiler compiler system. http://jastadd.cs.lth.se.

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

[EH06] Torbjörn Ekman and Görel Hedin. Modular name analysis for java using jastadd. In *Submitted for publication*, 2006.

[FFA99] Jeffrey S. Foster, Manuel Fahndrich, and Alexander Aiken. A theory of type qualifiers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, 1999.

[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.

[Hed00] Görel Hedin. Reference attribute grammars. In *Informatica (Slovenia)*, 24(3), 2000.

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

[Ing86] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In *OOPSLA*, pages 347–349, 1986.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *LNCS*, 2072:327–355, 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).

[MH03] Eva Magnusson and Görel Hedin. Circular reference attributed grammars - their evaluation and applications. *Electronic Notes of Theoretical Computer Science*, 82(3), 2003.

[Mil04] Todd Millstein. Practical predicate dispatch. In *OOPSLA*, 2004.

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

[SGSB05] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. *SIGPLAN Not.*, 40(10), 2005.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[TE05] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.