Rewritable Reference Attributed Grammars

design, implementation, and applications

Torbjörn Ekman

Licentiate Thesis, 2004

Department of Computer Science
Lund Institute of Technology
Lund University

ISSN 1652-4691
Licentiate Thesis 3, 2004
LU-CS-LIC:2004-3

Thesis submitted for partial fulfillment of
the degree of licentiate.

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

Email: torbjorn.ekman@cs.lth.se
WWW: http://www.cs.lth.se/~torbjorn

Typeset using LaTeX 2ε

Printed in Sweden by Media-Tryck, Lund, 2004

Abstract

This thesis 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 by means of attributes. We have implemented ReRAGs in our aspect-oriented compiler compiler tool JastAdd II. We present the ReRAG formalism, its evaluation algorithm, and examples of its use.

JastAdd II 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. Each technique is presented using a series of simplified examples from static semantic analysis for the Java programming language.

A case study is presented where ReRAGs are used extensively to implement a compiler for the Control Module extension to the IEC61131-3 automation languages. That Control Modules concept is further extended, in a modular fashion, with object-oriented features to improve encapsulation, composition mechanisms, code re-use, and type safety.

Preface

This thesis is for the Licentiate degree which is a Swedish degree between the MSc and PhD. It consists of an introductory part and three 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, 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.

III. Torbjörn Ekman, Design and Implementation of Object-Oriented Extensions to the Control Module Language. *Submitted for publication.*

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 three supervisors: Professor Boris Magnusson, Dr. Klas Nilsson, and Dr. Görel Hedin. I am particularly grateful to Görel Hedin who introduced me to reference attribute grammars. Much of the work on ReRAGs has been carried out jointly 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.

I would like to thank Anders Nilsson for various compiler related discussions and for using my Java front-end in his Java2c compiler, Anders Ive for being a great roommate and his work on byte-code generation using ReRAGs, and Eva Magnusson for her initial implementation of JastAdd.

The are also quite a few people that I would like to thank for nice ideas, joint work, and discussions on various topics; Roger Henriksson and Sven Gestegård Robertz for various GC related discussions, Ulf Asklund and Lars Bendix for joint work on agile configuration management, and Christian Andersson for all those rewarding discussions on computer related topics and life in general.

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 Institute of Technology, funded by ABB Automation Technology Products and VINNOVA, the Swedish Agency for Innovation Systems.

Contents

III Design and implementation of object-oriented extensions to the Control Module language

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, metric tools, consistency checkers, etc. These tools are usually complex and building them from scratch requires a major effort. This thesis addresses the problem of how to build such tools much more easily, providing high-level concise ways of programming the tools, and providing good modularity, allowing a high degree of reuse between tools.

The similarities between different program-processing tools are obvious. 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 a modular 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 prospective is to be able to easily 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.

The main contribution of this thesis is a new specification formalism, Rewritable Reference Attributed Grammars (ReRAGs), that raises the programming level for processing programs, and which is highly modular, supporting reuse among language tool implementations. ReRAGs have been implemented in the tool JastAdd II as a domain specific language extension on top of the Java programming language. Three synergistic mechanisms are used for supporting modularization: *inheritance* for model modularization, *aspects* for concerns cross-cutting the class hierarchy, and *rewrites* that allow computations to be expressed on the most suitable model. This enables specifications to be written in a highly modular fashion which is important both for understanding the specification, for breaking it down into simpler steps, and for reuse.

The rest of this introduction is structured as follows. Section 1 gives an overview of the papers included in the thesis. Section 2 gives a background to the various techniques that have served as inspiration when designing ReRAGs. Section 3 summarizes the contributions and Section 4 concludes the thesis and discusses some future work.

1 Thesis overview

This section gives a brief overview of the three papers included in this thesis.

1.1 Paper I: Rewritable Reference Attributed Grammars

This paper presents the ReRAG formalism, its evaluation algorithm, and examples of its use. Our largest application is a complete static-semantic analyzer for Java 1.4. Initial measurements using a subset of the Java class library as 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.

1.2 Paper II: A case study of Separation of Concerns in Compiler Construction using JastAdd II

This paper presents a case study of separation of concerns in compiler construction using the JastAdd II compiler compiler [Ekm04]. The ReRAG mechanisms of inheritance, aspects, and rewrites all support separation of concerns. This is illustrated through a series of simplified examples from static semantic analysis for the Java programming language.

1.3 Paper III: Design and Implementation of Object-Oriented Extensions to the Control Module Language

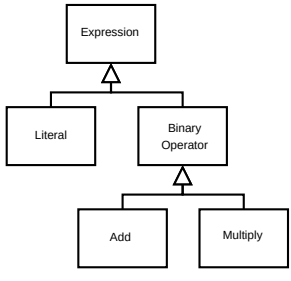
This paper presents a case study on language extensibility using ReRAGs. A DSL for programmable logic controllers is extended with language constructs that improve encapsulation, code reuse, and type safety. The paper presents both the design and the implementation of the language extensions.

2 Background

ReRAGs is a specification formalism that is inspired by several well known software development techniques: object-orientation, abstract syntax trees, aspect-oriented software development, declarative programming, attribute grammars, and rewriting systems. This section gives a brief introduction to these techniques.

2.1 Object orientation

Object orientation provides a natural way of structuring data and thinking about 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. Common behavior can be placed in an abstract superclass, e.g. *Expression*, and then inherited to the concrete subclasses. Overriding allows specialization of the inherited behavior if necessary. Object orientation provides a nice modularization mechanism through the class concept that models both state and behavior in a single module.

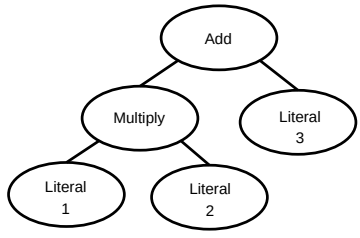


2.2 Abstract syntax trees

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. 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. The figure below shows a sample AST for the expression *1*2+3* built from objects in the class hierarchy described above.



2.3 Aspect oriented software development

Object orientation provides a classification hierarchy that allows us to modularize both state and behavior of a single language element. However, certain computations tend to cross-cut this classification hierarchy, e.g., evaluating the expression modeled by the AST above. To evaluate the expression it is natural, from an object-oriented perspective, to add a small piece of code in each class that performs the evaluation of that particular type. The computation would therefore cross-cut the entire class hierarchy.

Aspect oriented software development (AOSD), supported by tools such as AspectJ [KHH+01], deals with modularization and separation of concerns for features that cross-cut the main class hierarchy. It introduces new modularization concepts that allows state and behavior to be extracted from the main class hierarchy into modules that form a separate type hierarchy. A computation, such as evaluating an expression, can thus be separated into a single module instead of being scattered over multiple modules.

Within plain object-oriented programming without support for AOSD, the Visitor pattern [GHJV95], is often used to modularize such crosscutting computations. However, AOSD is much more powerful than the Visitor pattern, and provides a more direct and safer way to program such computations.

2.4 Declarative programming

While general purpose object oriented programming is usually 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, in turn, are made up of sets of equations describing the relations that specify what is to be computed. Declarative constructs thus only express what to be computed and not in which order it is to be computed. The actual order of the computations are then decided by the compiler. The combination of object oriented programming and declarative constructs can be very fruitful, both easy to work with and useful for large practical problems as will be described later in this thesis.

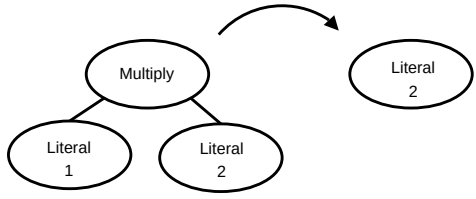
2.5 Attribute grammars

An attribute grammar (AG) supports declarative AST computations by decorating an AST with attributes that are defined by equations. An evaluator automatically solves the equation system for any AST following the base abstract grammar. Typical applications include the program analysis that is part of the front end of a compiler, e.g., name analysis 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]. Later developments within attribute grammars include the development of Higher Order Attribute Grammars (HAGs) [VSK89] that allow attributes to be new ASTs, with applications in, e.g., macro processing.

Reference Attributed Grammars (RAGs) are AGs augmented with the possibility for attributes to be links 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].

2.6 Rewrite systems

There are several systems based on AST rewriting, i.e., where rules are used for specifying how to transform a part of an AST to another form. Typical applications are code optimizations and reenginering of program source code. Influential systems include ASF+SDF [vdBea01] and Stratego [Vis01b] which are both based on algebraic specifications, and TXL [Cor04] which is based on functional programming. The figure below shows a rewrite that evaluates the expression and transforms the AST into the resulting value.



3 Contributions

The main contribution of this thesis is the development of the ReRAGs specification formalism. The formalism has been implemented in the JastAdd II tool and used to implement several langauge applications, including the full static semantics of the Java programming language. This section describes the contributions of this thesis from a ReRAGs design, implementation, and application perspective.

3.1 Design

We have introduced a technique for declarative rewriting of attributed ASTs, supporting conditional and context-dependent rewrites during attribution. 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 only be decided *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.

ReRAG specifications are highly modular supported by three synergistic mechanisms for separation of concerns. Inheritance separates the description of general behavior from specific behavior of the language constructs, e.g., general declarations from specialized declarations like fields and methods. Aspects separate different computations from each other, e.g., type checking from name analysis. Rewriting allows the computations to be expressed on the most suitable forms of the tree.

We have also identified several typical ways of transforming an AST that have proven useful in practice: Semantic Specialization, Make Implicit Behavior Explicit, and Eliminate Shorthands. These transformations have substantially simplified the implementation of an actual Java compiler in JastAdd II compared to having to program this by hand, or having to use a plain RAG on the inital AST constructed by a parser. Their use have been demonstrated by concrete examples from challenging static-semantic analysis computations for the Java programming language.

3.2 Implementation

The ReRAG technique has been implemented in our compiler compiler tool JastAdd II that generates compilers from a declarative specification. Attribute evaluation and tree transformation are performed automatically according to the specification. The rewrite and attribute evaluation engine allows partial caching of attributes. This caching is essential in gaining acceptable performance, as shown in [EH04]. The implementation has then been benchmarked using a ReRAGs specification of a full static-semantic analyzer for Java. 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.

3.3 Applications

ReRAGs have been used as the main language specification formalism in several research compilers and master thesis projects. The largest applications are described below:

Full Java static-semantics checker The largest application we have written is a complete static-semantic analyzer for Java 1.4. The grammar is a highly modular specification that follows the Java Language Specificaton, second edition [GJSB00], 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, is used to build the initial abstract syntax tree. The AST is then 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 *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] [NEN02] [NE01], using an older version of the Java checker as a front end. 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.

Extended Control Modules The domain-specific Control Module language used to implement programmable logic controllers is extended with object-oriented features to improve encapsulation, code reuse, and type safety. Common object-oriented features from general purpose languages are tailored to suit the modularization concepts and run-time model of the Control Module base language. The AST in the extended language is then translated into the base language. That tree is in turn exported to the exisiting ControlModule development environment and can thus benefit from existing infrastructure such as native compilers and run-time system.

Modular Tiger In a master's thesis project, a typechecker was developed for Andrew Appel's toy language Tiger, and extensions for object-orientation and generics were added as separate modules [San04].

Modular Java 1.5 extensions An ongoing master's thesis project investigates if it is possible to extend Java 1.4 in a modular fashion with the Java 1.5 extensions, e.g. generics, autoboxing, static imports, and an enhanced for-statement.

Finally, the JastAdd II tool have been implemented using ReRAGs and bootstrapped in itself.

4 Conclusions and future work

This thesis presents the ReRAGs specification formalism including its design, implementation, and various applications. The formalism raises the abstraction level for grammar-based computations using a declarative object-oriented technique to rewrite attributed abstract syntax trees. The rewrites support conditional and context-dependent rewrites. ReRAGs have been combined with aspect-oriented techniques in the JastAdd II tool that has been used to implement complex complete programming languages. ReRAGs support three synergistic mechanisms for separation of concerns: inheritance for module modularization, aspects for concerns cross-cutting the class hierarchy, and rewrites

that allow computations to be expressed on the most suitable model. This results in a compiler compiler that enables a high degree of separation of concerns and modularity for compiler phases such as name binding, type checking, and code generation. ReRAGs have been used to implement numerous languages, the largest being a full static semantic analyzer for Java.

4.1 Future work

ReCRAGs We plan to combine ReRAGs with Circular Reference Attributed Grammars [MH03]. This would result in grammars that support both rewrites and fixed-point computations. We think this will provide a useful basis for declarative specification of many problems in static analysis.

Integrated Development Environment We plan to integrate the JastAdd II tool into the open integrated development environment Eclipse. Besides a language sensitive editor it would be interesting to investigate how to support source level ReRAGs debugging and automated unit-testing.

Pattern Language During the design of ReRAGs we discovered several useful transformational patterns, e.g. Semantic Specialization, Make Implicit Behavior Explicit, and Eliminate Shorthands. Our various language implementations also share common design of name binding and type checking modules. It would be interesting to evaluate all these techniques and work towards a pattern language for compiler construction using JastAdd II.

Optimizations Our Java compiler, generated using JastAdd II, is a few times slower that traditional handwritten compilers. It would be interesting to further refine current caching strategies and evaluation strategies in JastAdd II to optimize performance. We are also interested in profiling the generated compiler and try to remove bottlenecks in the compiler specification to achieve performance closer to handwritten compilers.

Dynamic AOSD Features JastAdd II currently only uses static aspect oriented features in the form of inter type declarations also known as static introductions. This allows us to modularize concerns that crosscut the main classification hierarchy. Since rewrites are triggered by a child visit, the rewrites can be seen as dynamic features similar to the pointcuts in AspectJ [KHH+01]. It would be interesting to further investigate this relationship and to introduce other dynamic pointcuts as well.

References

[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.

[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.

[EH04] Torbjörn Ekman and Görel Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004: 18th European Conference on Object-Oriented Programming*, 2004.

[Ekm04] Torbjörn Ekman. A case study of Separation of Concerns in Compiler Construction using JastAdd II. In *Proceedings of the Third AOSD workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, 2004.

[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.

[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.

[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).

[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-verlag.

[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-Verlag, 1964.

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

[NE01] Anders Nilsson and Torbjörn Ekman. Deterministic java in tiny embedded systems. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*. IEEE Computer Society, 2001.

[NEN02] Anders Nilsson, Torbjörn Ekman, and Klas Nilsson. Real java for real time - gain and pain. In *Proceedings of the international conference on Compilers,*

architecture, and synthesis for embedded systems, pages 304–311. ACM Press, 2002.

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

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

[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.

[San04] Martin Sandin. Extending a Type Checker with Object Orientation and Generic Typing in a modular way using Higher-Order Reference Attributed Grammars. Master's thesis, Department of Computer Science, Lund Institute of Technology, 2004.

[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.

[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-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.

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.

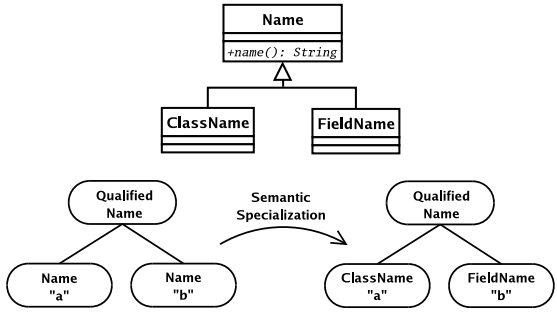


Figure 1. Semantic specialization of name references.

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.

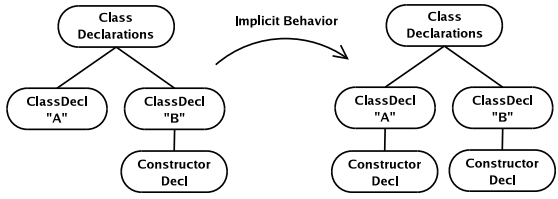


Figure 2. The implicit constructor in class “A” is made explicit.

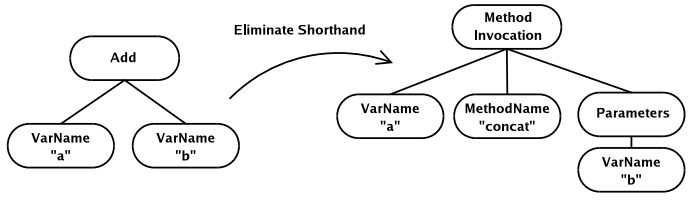


Figure 3. Eliminate shorthand and reflect the semantic meaning instead.

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.

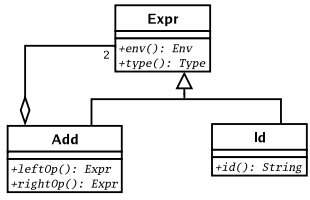
```
// Expr corresponds to a nonterminal
ast Expr;
// Add corresponds to an Expr production
ast Add extends Expr ::= Expr leftOp, Expr rightOp;
// Id corresponds to an Expr production, id is a token
ast Id extends Expr ::= <String id>;
```

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. □

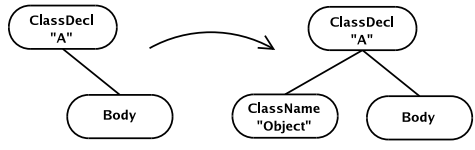
4.3 Rewriting descendent nodes

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'} ))
```

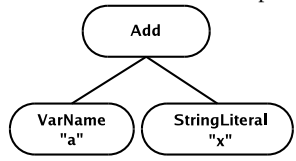
To simplify code generation we would like to eliminate the shorthand notation by rewriting the AST. This can be accomplished by a number of rewrite rules, each taking care of a single subproblem:

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

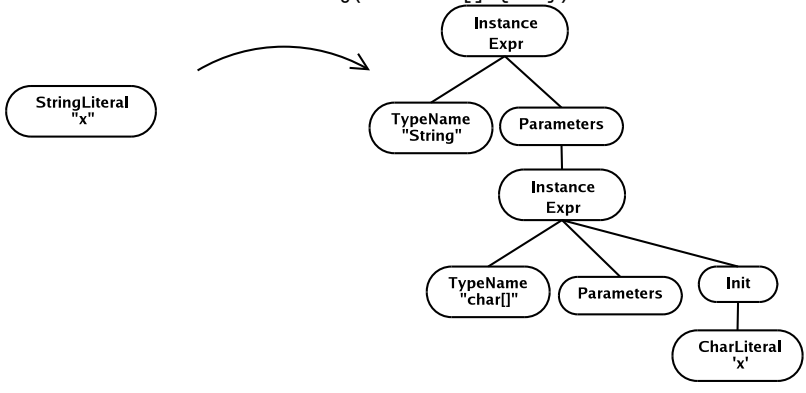
Suppose the original `Add` node is accessed from its parent. This will cause the AST to be rewritten in the following steps. First, it will be checked which rules are applicable for `Add`. This will involve accessing its left and right operands, which triggers the rewrite of these nodes in turn. In this 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.

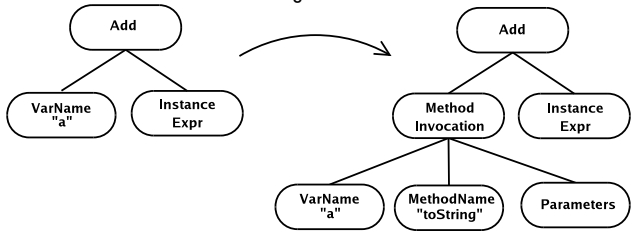
Initial AST for the `a + "x"` expression



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"`.

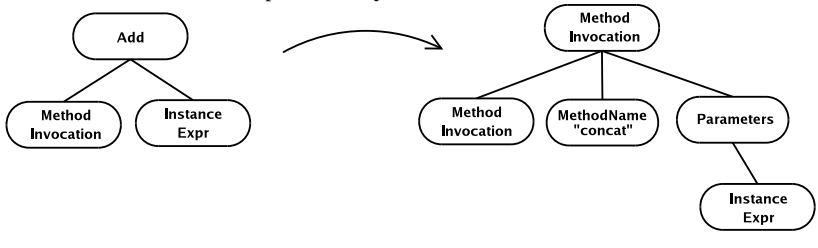


Figure 4. Combine several rules to eliminate the shorthand for String addition and literals in a Java like language.

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.

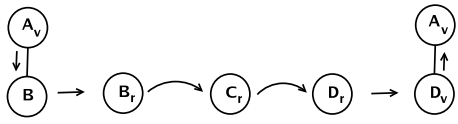
Attribute evaluation using this scheme will often follow complex tree traversal patterns, often visiting the same node multiple times in order to evaluate all the attributes that a specific attribute depends on. For example, consider the evaluation of the attribute *Id.type* above. This involves finding 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

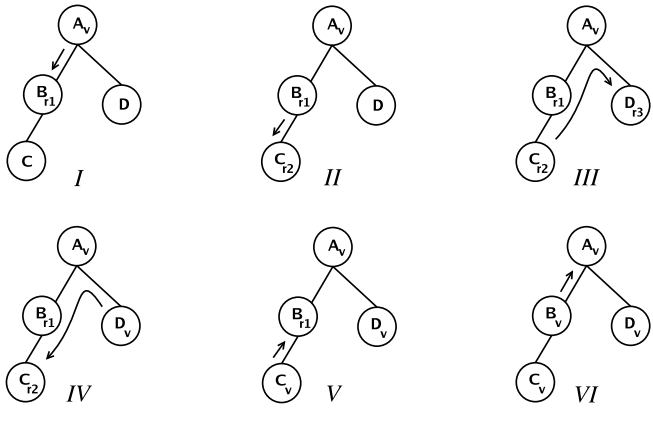
rewrite is currently being evaluated. When *B* is visited a rewrite is triggered and the node is rewritten to a *C* node that in turn is rewritten to a *D* node. No rewrite conditions for the *D* node are true, and the node is returned to the parent *A* that need not be aware of the performed rewrite.



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.

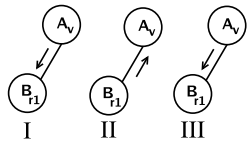
The following figure shows an example of nested rewrites. The subscript *v* indicates that a node has been visited and *r* that a rewrite is currently being evaluated. The rewrites are numbered in the order they started.



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*.




```
ast A ::= B child;
inh Type B.y();
eq A.child().y() = x();
syn Type A.x() = child().z();
syn Type B.z() = ... ;
```

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 reclassifi cation: 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

A case study of Separation of Concerns in Compiler Construction using JastAdd II

Torbjörn Ekman

Department of Computer Science, Lund University, Sweden
torbjorn.ekman@cs.lth.se

A case study of separation of concerns in compiler construction using JastAdd II

Torbjörn Ekman

Department of Computer Science, Lund University, Sweden
torbjorn.ekman@cs.lth.se

Abstract This paper presents a case study of separation of concerns in compiler construction using the JastAdd II compiler compiler. A domain-specific specification language, Rewritable Reference Attributed Grammars (ReRAGs), is combined with Java to implement compilers in a high-level declarative and modular fashion. Three synergistic mechanisms for separations of concerns are described: inheritance for model modularisation, aspects for cross-cutting concerns, and rewrites that allow computations to be expressed on the most suitable model. Each technique is presented using a series of simplified examples from static semantic analysis for the Java programming language.

1 Introduction

We present a case study of separation of concerns in compiler construction using the JastAdd II compiler compiler. Simplified examples from static semantic analysis for the Java programming language [GJSB00] are used to demonstrate the mechanisms for separation of concerns provided in JastAdd II. This work is part of a larger project where the entire static semantics of Java 1.4 have been implemented. We believe that Java is a suitable language implementation for experimenting on language modularisation because of its advanced scope rules with nested types and inheritance as well as need for reclassification of contextually ambiguous names during name analysis.

JastAdd II uses a declarative compiler specification in the form of Rewritable Reference Attributed Grammars (ReRAGs) [EH04] combined with imperative Java code. ReRAGs provide three synergistic mechanisms for separations of concerns: inheritance for model modularisation, 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.

The rest of this paper is structured as follows. Section 2 describes JastAdd II and its specification language. The three mechanisms for separation of concerns are demonstrated in sections 3, 4, and 5. Section 6 discusses how the

three mechanisms are used to deal with interaction between aspects. Section 7 points out some related work and Section 8 concludes this paper and discusses some future work.

2 JastAdd II Background

JastAdd II is an aspect-oriented compiler compiler tool using declarative Rewritable Reference Attributed Grammars (ReRAGs) and Java as its specification languages. The grammars define attributes and equations to specify computations and information propagation in the abstract syntax tree (AST). The formalism is object-oriented viewing the grammar as a class hierarchy and the AST nodes as instances of these classes. Behavior common to a group of language constructs can be specified in a common superclass and specialized or overridden for specific constructs in the corresponding subclasses. Often the most appropriate AST structure can only be decided after parial attribution of the AST. Rewrites allow restructuring of the tree to simplify the specification of the remaining attribution. The following sections give an introduction to JastAdd II compiler specifications.

2.1 The AST class hierarchy

The nodes in an Abstract Syntax Tree (AST) are viewed as instances of Java classes arranged in a subtype hierarchy similar to the one used in the Interpreter pattern, [GHJV95]. An AST class corresponds to a nonterminal or a production (or a combination thereof) and may define a number of descendents and their declared types, corresponding to a production right-hand side. In an actual AST, each node must be *type consistent* with its ancestor 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 ancestor. Shorthands for lists, optionals, and lexical items are also provided. All node types implicitly inherit the common ancestor type ASTNode that supports generic access to node children. This is particular useful for generic tree traversals. An example definition of some AST classes is shown below.

```
// Expr corresponds to a nonterminal
ast Expr;

// Add corresponds to an Expr production
ast Add : Expr ::= Expr leftOp, Expr rightOp;

// Id corresponds to an Expr production
// id is a token
ast Id : Expr ::= <String id>;
```

2.2 Reference Attributed Grammars

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.

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 an ancestor node. Note that the term *inherited attribute* refers to an attribute defined in the ancestor 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, unless explicitly stated otherwise.

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 ancestor 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.

In the JastAdd II implementation of RAGs attributes can be seen as methods where the method declaration and method body may be separated. Inherited attributes have their method body that defines the behavior in an ancestral node. An inherited attribute equation defines the behavior for a corresponding declaration of the same attribute in the subtree where the targeted equation node is the root. That way the only dependency on tree structure for that attribute is that the node holding the equation must be an ancestor to the node holding a declaration.

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 default equation for Add nodes
eq Add.type() = TypeSystem.INT;

// Overriding default equation for Id nodes
eq Id.type() = env().lookup(id()).type();
```

The notation for method invocation is used when accessing descendent 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.

2.3 Rewrite rules

ReRAGs extends RAGs with rewrite rules 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 ancestor 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.

A rewrite step is specified by a rewrite rule that defines the conditions when the rewrite is applicable, as well as the resulting tree. After the application of one rewrite rule, more rewrite rules may 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. Both the rewrite rule application order and the tree traversal order are implicitly defined by attribute dependencies. A thorough description of ReRAGs implementation and application will appear in [EH04].

3 Inheritance for model modularisation

The subtype hierarchy generated from the grammar production rules provide excellent support for model modularisation. Generic behavior is defined in the possibly abstract node types and then specialized in the concrete node types. A

small example adding a reference attribute to each expression referencing its corresponding type declaration node is shown below. The production rule hierarchy is in itself specialized in multiple steps, e.g binary operands, arithmetic expressions, and additive expressions are all successive specializations from the generic language element expression. The type reference is defined to be boolean for all relational types while the type of arithmetic expressions is the widest type of both operands. The approach is generic in the sense that adding another arithmetic expression, e.g. subtraction, does not affect type propagation but merely requires implementation of the unique behavior, e.g. code generation.

```
ast Expr ;
ast BinOp : Expr ::= Expr left, Expr right ;

ast ArithmeticExpr : Binop ;
ast AddExpr : ArithmeticExpr ;

ast RelationalExpr : Binop ;
ast LessThanExpr : RelationalExpr ;

syn Decl Expr.type() ;
eq ArithmeticExpr.type() =
    widestType(left().type(), right().type());
eq RelationalExpr.type() = TypeSystem.BOOLEAN;
```

4 Aspects for cross-cutting concerns

The examples shown so far are actually feature aspects where attributes that cross-cut the AST subtype hierarchy are grouped into separate modules. This technique is very similar to static introduction techniques used in AspectJ [KHH+01], Hyper/J [OT01], and Multi Java [CLCM00].

The example below is a simple name binding module that binds a use-site to its declaration site through the inherited attribute `bind` taking a name as its parameter. A block of statements is modeled as a list of statements and a list of declarations for simplicity. Each block introduces a new scope to search for declarations and there are nested scopes since each statement in a block can be a block itself. The inherited attribute `bind` must thus have an equation in each scope, i.e. the `Block` node, and if a matching declaration is not found the search must be delegated to the surrounding scope.

```
ast Block : Stmt ::= Stmt stmt*, Decl decl*;
ast Name : Expr ::= <String name>;
ast Decl ::= <String name>;
```

```
protected inh Decl Name.bind(String name);
protected inh Decl Block.bind(String name);

eq Block.stmt().bind(String name) {
  for(int i = 0; i < numDecl(); i++)
    if(decl(i).name().equals(name))
      return decl(i);
  return bind(name);
}

public syn Decl Name.decl = bind(name());
```

To limit coupling between aspects such as name binding and type checking it is useful to limit visibility of certain attributes outside the defining aspect. The only attribute that needs to be exported outside a name binding aspect is for instance the binding from a use-place to its declaration, e.g. `decl` in `Name`. Attributes that define scope rules, e.g. `bind`, only affect the name binding and should thus be private to name binding modules.

Aspects have proven a very powerful technique to implement design pattern roles, [HK02], [NK01]. The same technique can be used in JastAdd II to implement reusable modules, illustrated below where the name binding approach described above is generalized in a generic module for nested scopes. The involved actors are nodes that need to lookup declarations and nodes that define new scopes. These actors are specified as interfaces and later used to tag each tree node that takes the role of an actor defined in the module. These interfaces also specify the equations that the implementors must supply to define non-generic behavior, e.g. finding declarations in its scope that matches the provided name. In the example, `Scope` represents nodes that define a new scope and the non generic behavior is to match a name to a declaration while `Bind` represents the node that receives a reference to a declaration.

```
aspect NestedScopes {
  interface Scope {
    protected syn Decl lookup(String name);
  }

  interface Bind  {
    protected inh Decl bind(String name);
  }

  eq Scope.child().bind(String name) =
    lookup(name) != null ?
       lookup(name) : bind(name);
```

```
}
```

The module is generic in the sense that the only requirement on the AST structure is that an enclosing scope is defined by an ancestral node. It can be further generalized by adding more scope types, e.g. inheritance from super classes, and declare before use. Below is a name binding module that uses the module with the previously defined concrete node types `Block` and `Name`. The only behavior that needs to be implemented is the matching attribute `lookup` in `Block` and the use of the provided attribute `bind` in `Name`. In Java several nodes implement a scope, e.g. block, class, interface, and for statement, and thus share common properties.

```
aspect NameBinding extends NestedScopes {
  declare parents: Block implements Scope;

  declare parents: Name implements Bind;

  eq Block.lookup(String name) {
    for(int i = 0; i < numDecl(); i++)
      if(decl(i).equals(name))
        return decl(i);
    return null;
  }

  public syn Decl Name.decl = bind(name());
}
```

5 Rewrites to create the most suitable model

Rewrites can improve separation of concerns by allowing computations to be expressed on the most suitable model. The information acquired during the early stages of static semantic analysis can be used to rewrite the model to make that information explicitly visible in the model structure for later stages.

We use an example from Java name analysis to demonstrate the technique. When parsing an expression containing qualified names, e.g. `java.lang.System.out`, it is syntactically undecidable if a part of a name is a reference to a package, type, field, or variable unless their context is taken into account. In the above example, `java` is most often a package, but only as long as there is no variable-, field-, or type-declaration named `java` that would shadow the package according to the Java scope rules. Thus, a context-free grammar can only build generic name nodes that capture all cases. The attribution will need to handle all these cases and therefore becomes 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 ones. Other computations, like type checking, optimization, and code generation, can benefit from this rewrite by specifying different behavior in the specialized classes rather than having to deal with all the cases in the general name node.

An aspect that models Java names and resolves syntactically ambiguous names as described is shown below. There are two different types of names in Java from a syntactic point of view, simple names and qualified names. A simple name is a single identifier and a qualified name consists of a name, a "." token, and an identifier. During parsing a context-free grammar is used and thus general unbound names have to be built during AST creation. Semantic specialization is used to rewrite these general nodes into more specific ones, e.g. variable- or type-names. The ast-declarations in the aspect below model the described name structure.

Semantic specialization is implemented using a rewrite that rewrites an ambiguous `UnboundName` node into a `VariableName`-node or `TypeName`-node depending on the type of the binding received from the name binding module. Finally the `QualifiedName` nodes changes the scope rules for its right child to search the type of its left child to provide, e.g. when trying to bind out in the `System.out` expression the class `System` should be searched for a field named `out`.

```
ast Name : Expr;

ast SimpleName : Name ::= ID id;
ast QualifiedName : Name ::=
   Name left, SimpleName right;

ast UnboundName : SimpleName ;
ast VariableName : SimpleName ;
ast TypeName : SimpleName ;

// Resolve names depending on bound entity
rewrite UnboundName {
  when (bind().isVariableDecl())
  to SimpleName new VariableName(id());
  when (bind().isTypeDecl())
  to SimpleName new TypeName(id());
}

// The left name in a QualifiedName changes
// the scope for the name to the right
eq QualifiedName.right().bind(String name) {
  if(left() instanceof TypeName)
    return left().decl().lookup(name);
```

```
  if(left() instanceof VariableName)
    return left().type().lookup(name);
}
```

6 Aspect interaction

While the aspects demonstrated so far define static features we also use more pluggable aspects, e.g. a declare before use aspect to complement the name binding module in Section 4 and optional code optimization aspects. Pluggable aspects define rewrites that change a run-time node instance to a subtype node with extended behavior. That way an aspect can be added to the system in a way transparent to other aspects.

The examples demonstrated so far deal with equations that cross-cut the type hierarchy only and not cross-cutting concerns within equations. To override and extend attribute equations we use inheritance of the model structure in combination with rewrites that change the type of a node instance at run-time. I.e. we may have different/extended equations for UnboundName and VariableName defined in the example in Section 5. This technique, using run-time rewriting and inheritance, is more powerful than static compile-time point-cuts within equations in that it may take run-time information into account but less powerful in that each node may only be changed by a single aspect. Therefore it would be interesting to combine the current approach with more fine-grained static point-cuts within equations.

7 Related work

The introduction of attribute definitions and equations to an exisiting class hierarchy in a modular fashion used in JastAdd II is very similar to static introduction in AspectJ [KHH+01], hyperslices in Hyper/J [OT01], and open classes in MultiJava [CLCM00]. A functional approach to attribute grammar aspects using the same technique is presented in [dPJV00] where aspects are first-class objects that can be freely combined using a combinator library in Haskell. ReRAGs further improved modularisation support in that the current model instance may be rewritten during run-time to a more suitable model allowing each computation to be expressed on the most suitable model and more fine-grained separation of concerns within equations.

The Visitor pattern, [GHJV95], is often used in compiler construction for separation of concerns when using object-oriented languages. Visitors can only separate cross-cutting methods while static introductions can be used for fields

as well. AOP implementations of the Visitor pattern need not rely on a delegation mechanism resulting in a cleaner more intuative implementation, [HK02]. ReRAGs aspects differ from AOP implementations of the Visitor pattern in that an explicit traversal strategy in the form of a Visitor is not specified but merely implicitly defined by attribute dependences. Rewrites further improve modularisation in that the underlying structure may change during run-time to better fit the current computation.

Higher order attribute grammars (HAGs) [VSK89] allow trees as attributes that are defined as a function of the partially attributed existing AST at run-time and can thus provide a more suitable model. The process is, however, not transparent to other computations and is thus less flexible from a separation of concerns view. The use of attribute grammars and forwarding for modular language implementation is discussed in [VWMBK02]. Forwarding overrides attribute equation dynamically at run-time and forwards equation to a different part of the tree. Since it is based on HAGs the target tree can be computed at run-time and the approach is thus similar to semantic specialization.

8 Conclusions and future work

We have demonstrated three synergistic mechanisms for separations of concerns supported by ReRAGs in the JastAdd II compiler compiler: inheritance for model modularisation, aspects for cross-cutting concerns, and rewrites that allow computations to be expressed on the most suitable model. Examples inspired by static semantic analysis of the Java programming languages have been used to illustrate and motivate each technique. We believe that this allows compilers to be written in a high-level declarative and modular fashion.

Our experiences indicate that the implementation leads to flexible solutions to several traditional compiler construction problems, and we hope to generalize some of these techniques and document them as design patterns or frameworks for compiler construction using ReRAGs.

We would also like to investigate the interaction between pluggable aspects and also how to better support fine-grained cross-cutting within equations combining AspectJ-like point-cuts with run-time rewriting implemented using ReRAGs in JastAdd II.

References

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of OOPSLA 2000*, volume 35(10), pages 130–145, 2000.

[dPJV00] Oege de Moor, Simon Peyton-Jones, and Eric Van Wyk. Aspect-oriented compilers. *Lecture Notes in Computer Science*, 1799, 2000.

[EH04] Torbjörn Ekman and Görel Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004: 18th European Conference on Object-Oriented Programming*, 2004.

[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.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of OOPSLA-02*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 161–173, New York, November 4–8 2002. ACM Press.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 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).

[NK01] N. Noda and T. Kishi. Implementing design patterns using advanced separation of concerns. In *OOPSLA2001 workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[OT01] Harold Ossher and Petri Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *Proceedings of the 23rd international conference on Software engineering*, pages 821–822. IEEE Computer Society, 2001.

[VSK89] 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.

[VWMBK02] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In R. N. Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Grenoble, France, April 8-12, 2002*, volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer-Verlag, 2002.

Paper III

Design and implementation of object-oriented extensions to the Control Module language

Torbjörn Ekman

Department of Computer Science, Lund University, Sweden
torbjorn.ekman@cs.lth.se

Design and implementation of object-oriented extensions to the Control Module language

Torbjörn Ekman

Department of Computer Science, Lund University, Sweden
torbjorn@cs.lth.se

Abstract The Control Module language is a domain-specific language used to implement programmable logic controllers. This paper presents the design and implementation of a set of language extensions that improve encapsulation, code re-use, and type safety. The extensions as well as the base language are implemented using Rewritable Reference Attributed Grammars supported by the JastAdd II tool. Static semantic analysis and code generation are implemented in a modular declarative fashion using the abstract syntax tree as the only data structure. To better reflect the semantics of the language the tree structure is dynamically rewritten during compilation. The extensions are finally translated into the base language to re-use existing compilers and run-time system.

1 Introduction

This paper presents the design and implementation of object-oriented language extensions to the domain-specific Control Module language from ABB. Control Modules are used to implement programmable logic controllers (PLCs) and are in turn an extension to the IEC61131-3 standard [JT01].

Since this paper focuses on both the design of language extensions as well as the techniques used to implement them the contribution is twofold. The first contribution is a case study where a domain-specific language (DSL) is extended to benefit from popular features in general-purpose programming languages while preserving unique properties of the DSL such as execution model and run-time system. The second contribution is a detailed explanation of how the static semantics and code generation for the extended Control Module language can be expressed in a natural way using Rewritable Reference Attributed Grammars (ReRAGs) [EH04].

The Control Module language is extended to improve encapsulation, code re-use, and safety. Common object-oriented features from general purpose languages are tailored to suit the modularization concepts and run-time model of the domain-specific Control Module language. The extensions are motivated and demonstrated through a set of code examples.

The abstract syntax as well as the static semantics for the base language are modeled using ReRAGs in the JastAdd II tool. The description of this part serves as an introduction in how to write a compiler in JastAdd II with modularity and separation of concerns in mind. The extensions are then modeled separately in a similar manner with separate modules for name binding, type checking, and code generation. The implementation has been done with two important design principles in mind. The abstract syntax tree (AST) should be the only data structure, i.e. no external data structures such as symbol tables are used, and all program elements should be explicitly visible in the AST, i.e. all declarations including implicit elements such as primitive type declarations are included in the tree structure. Since all computations are performed on a single data structure it is important that the chosen structure suits various computations such as name binding, type checking, and code generation. This is achieved by dynamically rewriting the tree into a form suitable for the current computation phase. This also allows information gathered during a previous phase, e.g. the type of an expression, to be reflected in the tree structure. The final tree is then translated from the extended language into the base language. That tree is in turn exported to the existing ControlModule development environment and can thus benefit from existing infrastructure such as native compilers and run-time system.

The rest of the paper is structured as follows. Section 2 describes the Control Module language from a code organization and execution model perspective. Section 3 introduces some language extensions and describes how these features affect encapsulation and composition. ReRAGs and the JastAdd II tool are introduced in Section 4. Section 5 describes how the base language and the extended language features can be implemented in a modular fashion. Section 6 concludes the paper.

2 Control Modules

The Control Module concept from ABB Automation Technology Products is an extension to the the IEC61131-3 standard. The languages in IEC61131-3 are often used in Programmable Logic Controllers (PLCs). The early PLCs were intended to replace relay based control systems, but current PLCs are used for most types of control systems in all kinds of industry plants. These systems are real-time systems where the programs must respond to changes in the environment within a specified time limit. A typical program reads input signals, performs some computation using these values as well as historic data, and finally updates the output signals. This cycle is then repeated with a specified frequency. There are usually several concurrent programs that may execute with different cycle times.

ABB's control modules extend the traditional modularization facilities and execution model in the IEC61131-3 standard. A control module is a container for both application programs and graphical objects but this paper will only discuss the application aspect of the control module.

This section gives a brief introduction to both the IEC61131-3 languages and the Control Modules from a code organization and execution model perspective.

2.1 The IEC61131-3 standard

The IEC61131 standard is a specification for programmable controllers where IEC61131-3 specifies the syntax and semantics of a unified suite of programming languages. This section gives a very brief introduction to the standard from a code organization, communication model, and typing perspective. A detailed description of the standard is given in [JT01].

The standard describes five languages:

Instruction List (IL) Textual language similar to low-level assembly language.

Ladder Diagram (LD) Inspired by electrical wiring diagrams describing relay based control systems.

Structured Text (ST) Textual language with a Pascal-like syntax.

Function Block Diagram (FBD) Graphical language where complex functions are composed from a set of function blocks connected with flow signals.

Sequential Function Chars (SFC) Graphical language to describe sequential behavior that has evolved from Grafcet [TC79].

All languages have a static structure with no dynamic memory allocation and type instantiation. Since we mainly deal with code organization in this paper there will be very few examples with code. When needed, simple snippets of Structured Text will be used for illustration purposes.

Control applications are divided in *program organization units* (POUs). POUs may be nested to provide support for hierarchical structuring of the code. However, the standard states that POUs shall not be recursive in the sense that an invocation of a POU shall not cause the invocation of another POU of the same type. This ensures that the size of the code as well as the data structures in the POUs is statically computable at compile-time.

There are three different types of POUs:

Program Each declared program is a single instance that defines a set of input/output parameters, local variables, and a code block. The code block is implemented in one of the five previously described languages and executed each cycle. A program may also contain nested function blocks, and functions. Parameters in these nested POUs are initialized by the enclosing program using parameters, local variables, or constant values.

Function Block A function block declaration is a type declaration encapsulating state as well as behavior. The state is specified by input/output parameters and local variables. The behavior is specified by a code block implemented in one of the five described languages. Each POU using a specific function block must instantiate that function block and give it a unique name. That name is used later to explicitly invoke the function block. During invocation the input and output parameters must be bound to local variables or parameters. Function blocks support nested POUs in a way similar to programs except that function blocks may not contain nested programs.

Function A function is a stateless operation that must return the same value each time it is invoked with the same parameters. The function evaluates a code block and returns a single value and may thus be used as an expression in any of the five IEC61131-3 languages. The function does not support nested POUs.

2.2 ABB's Control Module language

The Control Module is an extension to the IEC61131-3 languages that introduces a fourth POU, the Control Module. Control modules are similar to Function Blocks in that they are instantiable types with both state and behavior. Control modules also support composition of POUs through nested modules, function blocks, and functions. The main difference between control modules and function blocks, except for the graphical objects, is a enhanced execution model. This section describes the Control Modules language through a series of examples to illustrate the code organization, execution model, and typing facilities supported by the language.

A first example defining a Control Module named *consumer* is shown below. The consumer has a need to consume a certain amount of an unspecified entity. This amount is specified by the *need* parameter. There is a producer that offers the consumer the amount specified in *offer* to be consumed each scan. There is, however, a *limit* in how much the consumer can consume each execution scan. Therefore, the actual consumed *quantity* by the consumer during this scan is set to be the lowest value of the limit and the offered request.

```
CONTROLMODULETYPE CONSUMER
  VAR_IN_OUT
    need : INT;     // Total need
    quantity : INT; // Consumed during the latest scan
    offer : INT;    // Offered amount during current scan
  END_VAR_IN_OUT

  VAR
```

```
    limit : INT := 5; // Limits the amount to
  END_VAR             // consume each scan

  CODEBLOCK EVAL    // Executed once each scan
    quantity := MIN(limit, offer);
  END_CODEBLOCK
END_CONTROLMODULETYPE
```

The *consumer* module contains both state and behavior where the state is implemented by parameter variables and local variables while the behavior is implemented in a single code block named *eval*. The module has three parameters that are initialized when the control module is instantiated. The reason that the *need* variable is supplied as a parameter instead of a local variable as the *limit* variable will be explained in the next section. The parameters are passed by name and the change of the *quantity* parameter may thus affect other variables bound to the same entity as that parameter. There is also a local variable *limit*, initialized at the declaration site, that can not be accessed outside the module.

Communication model Control Modules communicate with each other by exchanging data through parameters. The reason for this communication scheme is that a variable is only accessible from the program organization unit where it is declared, but can be shared with a nested POU through a parameter. Since parameters are call by name the same variable can be assigned in any nested POU that receives the variable as a parameter. If there are two POUs that have a sibling relationship but still need to share data, that common data must be placed in a common ancestor and propagated to both POUs through parameters.

Consider the example below that extends the before mentioned *Consumer* example. A new control module *Producer* is added that is responsible for producing the unspecified entity that the consumer needs. The producer offers the consumer the amount found when subtracting the already delivered amount from the total needed amount. The delivered amount is then updated to include the quantity actually consumed by the consumer.

```
CONTROLMODULETYPE PRODUCER
  VAR_IN_OUT
    delivered : INT;
    offer : INT;
    quantity : INT;
    need : INT;
```

```
  END_VAR_IN_OUT

  CODEBLOCK
    offer := need - delivered;
    delivered := delivered + quantity;
  END_CODEBLOCK
END_CONTROLMODULETYPE
```

The producer and consumer need to communicate data and that can only be done through common parameters, e.g. *delivered*, *offer*, *quantity*, and *need* in this case. This results in a situation where data whose state is defined by a module (and therefore logically belongs to the module) cannot be declared in that module, but needs to be passed around as a parameter. It would be desirable to encapsulate that data in the module instead of in an unrelated module that happens to be an ancestor node in the composition hierarchy. In the example the variables *delivered* and *offer* logically belong to the *Producer* while *quantity* and *need* logically belong to the *Consumer* module. But since both modules need to access the data, the variables must be located in a common ancestor module and passed to the Consumer and Producer as parameters. This is illustrated by the *System* module below. Furthermore, the *need* parameter is initialized in the System module instead of in the consumer module for which it defines the state. The example also illustrates hierarchical composition through the instantiation of the *Producer* and *Consumer* control module types.

```
CONTROLMODULETYPE SYSTEM
  VAR
    delivered : INT;
    offer : INT;
    quantity : INT;
    need : INT := 100;
  END_VAR

  CONTROLMODULE
    prod : PRODUCER ( deliviered, offer, quantity, need );
    con : CONSUMER ( need, quantity, offer );
  END_CONTROLMODULE
END_CONTROLMODULETYPE
```

There are also global variables that can be used to propagate information between any program or function blocks. This leads, however, to very poor encapsulation and is not further discussed in this paper.

Execution model The main difference between control modules and function blocks, except for the graphical objects, is the execution model. While function blocks are explicitly invoked, the execution model for control modules is partly declarative in that it is based on static data flow analysis. The order that the code blocks are executed is thus not explicitly specified in the code but evaluated at compile-time taking data dependences into account. The goal is to allow data that is modified during one execution scan to be available during the following scan. The flow analysis is global in the sense that all code blocks scheduled on the same task are sorted as a group to find a suitable order. Each code block is then executed sequentially in the pre-determined order each execution scan.

Control modules may have multiple code blocks that enable bi-directional communication between POUs during a single scan. In the *Producer-Consumer* example the code block in the *Producer* can be divided into two separate parts. The first part calculates how much to offer the consumer and the second part updates the amount actually consumed. If a single code block is used in the consumer, the update part will read the amount from the previous scan instead of the current scan. However, if the producer is implemented as shown below the compute block in producer can be executed first, followed by the block in consumer, and finally the update block in producer, resulting in an execution order where all values that are used in the computation are the values computed during the current scan.

```
CONTROLMODULETYPE PRODUCER
  VAR_IN_OUT
    delivered : INT;
    offer : INT;
    quantity : INT;
    need : INT;
  END_VAR_IN_OUT

  CODEBLOCK COMPUTE
    offer := need - delivered;
  END_CODEBLOCK
  CODEBLOCK UPDATE
    delivered := delivered + quantity;
  END_CODEBLOCK
END_CONTROLMODULETYPE
```

Type system The IEC61131-3 standard defines both primitive data types and user defined structured data types. The primitive types are traditional types like boolean, signed and unsigned integers of various lengths, floating point types,

etc. There are also some generic primitive data types, e.g. all integers, that can be used to define functions with overloaded parameters that operate on several data types. There is no corresponding type hierarchy for the user-defined structured data types. These form a flat type system with no type substitution.

Control Modules and Function Blocks are user defined types that may be instantitated. The instantiation is, however, the only operation that take the type of these entitites into account. There is no way to specify how two modules relate to each other from a type perspective, only what kind of data each module can operate on. It is thus possible to verify, at compile-time, that two modules are connected through parameters that are of the correct type, but there is no way to ensure that the two module types are compatible. In the *Producer-Consumer* case we can verify that the parameters that are connecting the two modules to each other are correctly typed, but not that the *Producer* is actually connected to a *Consumer* module. The possibility to express these types of relations between modules would improve the compile-time safety of the system. This could be further enhanced by introducing type hierarchies with type substitutability where the *Consumer* type constraint could represent a whole class of types.

3 Extended Control Modules

This section presents extensions to the Control Module concept to enhance encapsulation, composition mechanisms, code re-use, and static analysis for better safety. The extensions are heavily influenced by strongly typed OO languages such as Simula [DMN68], Beta [MMPN93], and Java [GJSB00], but tailored to ensure that the static nature of Control Modules is preserved. That implies that only static instantiation of modules is allowed and that there is no dynamic allocation of data structures. The extensions also preserve the declarative communication model and the code block sorting used in Control Modules.

All extensions are expressible in the base language, and code generation is implemented by transforming into the base language. The extensions can thus be combined with other POUs in IEC61131-3 and also benefit from existing infrastructure such as compilers and run-time systems. The code generation is described in Section 5.

In the following discussion we will use terminology from object-oriented languages to describe the extended Control Modules. A module type corresponds to a class and local variables declared in a module correspond to instance variables. Control modules that are instantiated within another Control Module are part objects of that module.

3.1 References to access remote control modules

References are added to provide non local access between Control Modules. A module may have a reference to another control module and access remote instance variables by dereferencing a reference and access the variables directly in the target control module. A reference can either be a remote reference passed around the tree as a parameter or a local part object reference that holds the reference to instance part objects.

References are single assignment variables that may only be assigned once and that is during the creation of a control module. The references are thus declarative in the same sense as parameters. References may, however, be dereferenced multiple times in any desired code block.

References improve encapsulation and strengthen the composition of modules. To access data from multiple module instances, using traditional Control Modules, the variables that need to be accessed must to be located in an ancestor node common to all involved modules. This leads to an encapsulation problem where the data can not be located in the logical module but must change location depending on the actual application's module composition. The variables are then explicitly propagated down the subtree where the ancestor declaring the variables is the root node. Even if the propagation can be somewhat simplified by grouping several variables into a single data type, modules in the subtree that are not accessing the variables must propagate the data down to children accessing those variables. By allowing references for non local access, the data can be placed in the conceptually logical module and still be accessed by remote modules, thus leading to better encapsulation.

The *Consumer* control module implemented with references is shown below. Data that belongs logically to the *Producer* is placed directly in the *Producer* and accessed through remote access by dereferencing the *prod* reference instead of through parameters. The *System* module need not hold the common data as local variables but can merely provide references between the producer and consumer when they are being instantiated.

Since local variables can be accessed from a remote location through reference access, the *limit* variable that used to be private to the Consumer is now visible to all modules that have a reference to the Consumer. This problem is further discussed in 3.4.

```
CONTROLMODULETYPE CONSUMER
  VAR_IN
    prod : PRODUCER;   // Reference
  END_VAR_IN

  VAR
```

```
    limit : INT := 5;  // Instance variable
    quantity : INT;    // Instance variable
    need : INT := 100; // Instance variable
  END_VAR

  CODEBLOCK EVAL       // Dereferenced reference
    quantity := MIN(limit, prod.offer);
  END_CODEBLOCK
END_CONTROLMODULETYPE

CONTROLMODULETYPE PRODUCER
  VAR_IN
    con : CONSUMER;   // Reference
  END_VAR_IN

  VAR
    delivered : INT;  // Instance variable
    offer : INT;      // Instance variable
  END_VAR

  CODEBLOCK COMPUTE   // Dereferenced reference
    offer := con.need - delivered;
  END_CODEBLOCK

  CODEBLOCK UPDATE   // Dereferenced reference
    delivered := delivered + con.quantity;
  END_CODEBLOCK
END_CONTROLMODULETYPE

CONTROLMODULETYPE SYSTEM
  CONTROLMODULE
    prod : PRODUCER ( con ); // Part object
    con : CONSUMER ( prod ); // Part object
  END_CONTROLMODULE;
END_CONTROLMODULETYPE
```

3.2 Type System

The type system is improved with a subtype relationship for control module types and not only for primitive types. A subtype must implement the same interface as the supertype but may add additional specifications as well. The

interface for a control module type is the entities of that type that are visible to another module, e.g. part object references, references variables, and instance variables. The subtype relationship is specified in the Control Module Type declaration explicitly by letting the new type extend an existing type and that way become a direct subtype of that type.

Remote instance references have a static qualified type that constrains the instance of the module that the reference binds to, i.e. the target module must be a subtype of the static qualified type. When accessing a module remotely through a reference only those properties that are visible in the static type are accessible when dereferencing that reference. This improves encapsulation during composition since the target module can be replaced by any subtype of the reference's static qualified type without changing any code that uses that reference.

The example below implements a *SubConsumer* that is a subtype of the *Consumer* type. When the producer is being instantiated the reference *con* refers to a *SubConsumer* instead of the *Consumer* used in the previous examples. Since *SubConsumer* is a subtype of *Consumer* the argument is valid and the producer is not affected by the changed consumer.

```
CONTROLMODULETYPE SUBCONSUMER EXTENDS CONSUMER
  ...
END_CONTROLMODULETYPE

CONTROLMODULETYPE SUBSYSTEM
  CONTROLMODULE
    prod : PRODUCER ( con );    // Part object
    con : SUBCONSUMER ( prod ); // Part object
  END_CONTROLMODULE;
END_CONTROLMODULETYPE
```

3.3 Inheritance

The type system is also extended with inheritance to re-use specification and implementation from a supertype module. When a control module inherits a supertype the module becomes a subtype of that supertype and automatically receives the same specification and implementation as the supertype. The inherited elements are the type, remote references, part objects references, instance variables, and code blocks. This is not only a powerful way to extend type hierarchies but also avoids duplication of code.

Inheritance for behavior Inheriting the default behavior in a subtype from the supertype is a nice way to avoid duplication of code, but often the behavior needs to be specialized in the subtype. The extended control modules provide two different techniques to specialize code blocks, *overriding* and *extension*. When overriding a code block the implementation in the supertype is discarded and the entire implementation is provided in the subtype. The *SubConsumer* can be implemented, as shown below, by extending the *Consumer* and overriding the *eval* code block. This particular implementation can consume unlimited amounts and therefore accepts any offer from the producer. The developer is free to override the code block in the supertype with an arbitrary code block written in any IEC61131-3 language, e.g. Structured Text. It is thus up to the developer to ensure that the *SubConsumer* can replace any *Consumer* to allow type substitutability.

```
CONTROLMODULETYPE SUBCONSUMER EXTENDS CONSUMER
  OVERRIDE CODEBLOCK EVAL
    quantity := prod.offer; // Dereferenced reference
  END_CODEBLOCK
END_CONTROLMODULETYPE
```

When a code block is extended, the implemented behavior in the supertype is extended with additional behavior implementation in the subtype. The supertype is responsible for defining where the additional implementation should execute within the sequential code block using the keyword *inner* [DMN68] [MMPN93]. The additional behavior specified in the subtype is then executed sequentially when *inner* is reached in the supertype. This is particularly useful when the existing implementation must ensure some minimal behavior. The example below declares an abstract ControlModuleType *DeltaConsumer* that tracks the change of the accepted quantity each scan in the *delta* variable. The type is abstract since the *Eval* code block has an *inner* statement that requires extension in a subtype. The *LimitedConsumer* extends the *eval* code block and updates the *quantity* variable. The supertype thus defined some minimal start and end behavior that is then specialized in the subtype.

```
CONTROLMODULETYPE DELTACONSUMER EXTENDS CONSUMER
  VAR
    delta : INT; // Instance variable
    temp : INT;  // Instance variable
  END_VAR

  CODEBLOCK EVAL
    temp := quantity;
```

```
    inner;
    delta := quantity - temp;
  END_CODEBLOCK
END_CONTROLMODULETYPE

CONTROLMODULETYPE LIMITEDCONSUMER EXTENDS DELTACONSUMER
  EXTEND CODEBLOCK EVAL
    quantity := MIN(limit, offer);
  END_CODEBLOCK
END_CONTROLMODULETYPE
```

Inheritance for composition Inheritance in combination with overriding is not only useful to specialize behavior but also to specialize hierarchical composition as noted in [MMPN93]. The extended Control Modules allow the inherited declaration of a part object to be overridden with a part object declaration that is a subtype of the type declared in the extended control module, i.e., co-variant part object specialization. Co-variant types have been the subject of much debate within the object-oriented community because they may cause the need for run time type checks in certain situations when references are passed as parameters to methods. However, this is not a problem here because references are used only in static configurations of Control Modules, and are never passed as parameters to methods.

This co-variant part object specialization turns out to be a powerful way to specialize the hierarchical composition of a Control Module Type. The implementation of *SubSystem* below extends the *System* module and overrides the *con* part object to be a *SubConsumer*. This is in effect the same module as the changed system in Section 3.2 when implemented using specialized composition.

```
CONTROLMODULETYPE SUBSYSTEM EXTENDS SYSTEM
  CONTROLMODULE
    con : SUBCONSUMER ( prod ); // Part object
  END_CONTROLMODULE;
END_CONTROLMODULETYPE
```

When a module accesses instance references and parameter references in a remote module through dereferencing a reference, those two types of references are identitcal from the accessors viewpoint. The local part object reference is accessed in the same way as a parameter to a remote instance. A local part object reference in a supertype may thus be overridden with a remote instance reference parameter in a subtype and vice versa. This further

improves the composition mechanisms when specializing control module type hierarchies. The example below specializes the system in that the consumer is accessed through a parameter instead of a part object instantiated in the *System* module.

```
CONTROLMODULETYPE SUBSYSTEM EXTENDS SYSTEM
  VAR_IN_OUT
    con : SUBCONSUMER; // Remote reference parameter
  END_VAR_IN_OUT
END_CONTROLMODULETYPE
```

3.4 Protection mechanisms

The new features introduced in control modules may be too flexible in certain situations and therefore a protection mechanism inspired by Java [GJSB00] is added. This ensures that the developer can put limitations on how to specialize a module and also ensure that not too much of the internal implementation is exposed to other modules.

The implementation status of each code block declaration can be set to abstract or final when declared. An abstract entity is not implemented in the module but must be implemented in each concrete control module extending the supertype behavior. A module containing abstract code blocks may not be instantiated and an attempt to do so will result in a compile-time error. This allows generic behavior to be located in an abstract type while ensuring that concrete behavior is provided before instantiation. A final code block indicates that the implementation is final and may not be overridden or extended in any subtype. This is also verified at compile-time.

There are three protection levels that define which parts of a module implementation are exposed to other modules. These levels apply to all entities that may be reached from a remote module, i.e. parameters and local variables. A public entity may be accessed through remote module invocation from any other module, a protected entity may only be accessed locally from the declaring module and its subtypes, and a private entity may only be accessed from the declaring module.

The example below shows an enhanced *Consumer* implementation that uses the protection mechanisms. As discussed in 3.1. the introduction of references made the *limit* variable visible to other modules. The *protected* keyword makes that variable visible only to the declaring module and its subtypes and may thus be overridden but not accessed from other modules. The reference to the producer is considered *private* to the declaring module and may not be accessed from other modules. The computation of the *quantity* attribute is re-

quired for the module and possible submodules and therefore declared *final* to ensure that it is not extended nor overridden.

```
CONTROLMODULETYPE CONSUMER
  VAR_IN
    PRIVATE prod : PRODUCER;     // Reference
  END_VAR_IN

  VAR
    PROTECTED limit : INT := 5;  // Instance variable
    quantity : INT;              // Instance variable
    need : INT := 100;           // Instance variable
  END_VAR

  FINAL CODEBLOCK EVAL           // Dereferenced reference
    quantity := MIN(limit, prod.offer);
  END_CODEBLOCK
END_CONTROLMODULETYPE
```

3.5 Language extension summary

This section has described some language extensions to the Control Module language. The extensions have been designed to provide better encapsulation, promote code re-use and improve the static compile-time safety. The following extensions, inspired by object-oriented techniques but tailored to fit the control module language, have been proposed:

- References for remote access between control modules
- Hierarchical type system with type substitutability
- Inheritance to promote re-use and specialization of behaviour and composition
- Protection mechanisms to limit the externally exposed interface to a module

Each extension has been demonstrated using small code examples and its effect on the above described design criteria is motivated. The examples so far have been very small and mainly used to describe the new language features. It would be interesting to evaluate the extensions on some realistic examples and compare the result to existing designs. We leave a thorough evaluation of the extended control modules as future work and go on to discuss the implementation.

4 JastAdd II Background

This section gives an introduction to the JastAdd II tool used to implement the language extensions proposed in the previous section. JastAdd II [Ekm04] is an aspect-oriented compiler compiler tool using declarative Rewritable Reference Attributed Grammars (ReRAGs) and Java as its specification languages. The grammars define attributes and equations to specify computations and information propagation in the abstract syntax tree (AST). The formalism is object-oriented, viewing the grammar as a class hierarchy and the AST nodes as instances of these classes. Behavior common to a group of language constructs can be specified in a common superclass and specialized or overridden for specific constructs in the corresponding subclasses. Often the most appropriate AST structure can only be decided after partial attribution of the AST. Higher order attributes allow new parts of the tree to be defined as a function of the existing, partially attributed tree. These attributes are higher-order in that the new created trees may contain new attributes. Rewrites allow restructuring of the tree to simplify the specification of the remaining attribution. The following sections give an introduction to JastAdd II compiler specifications.

4.1 The AST class hierarchy

The nodes in an Abstract Syntax Tree (AST) are viewed as instances of Java classes arranged in a subtype hierarchy similar to the the hierarchy used in the Interpreter pattern [GHJV95]. An AST class corresponds to a nonterminal or a production (or a combination thereof) and may define a number of descendants and their declared types, corresponding to a production right-hand side.

The ControlModuleType declaration below defines a ControlModuleType AST node that has a list of children where each child is an instance of the Member type. The Member production corresponds to a nonterminal while CodeBlock, Parameter, PartObject, and LocalVariable correspond to a Member production.

```
ast ControlModuleType ::= Member member*;
ast Member ::= <String name>;
ast CodeBlock : Member ::= Stmt stmt*;
ast Parameter : Member ::= <String typeName>, Expr expr;
ast PartObject : Member ::= <String typeName>, Expr arg*;
ast LocalVar : Member ::= <String typeName>, Expr expr;
```

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. A CodeBlock node or any other node that is a production of Member can thus be a child of the ControlModuleType node.

The star indicates that there is a list of elements while the angle brackets indicate an lexical item that should be initialized at node construction time, e.g., initialize an identifier with the actual token. Each element on the right hand side of a production has a name that is used to access that element from the parent node. E.g., the arguments of the PartObject are nodes of the Expr type that can be accessed using the arg name from the PartObject node.

4.2 Reference Attributed Grammars

ReRAGs are based on Reference Attributed Grammars (RAGs) [Hed00] 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. 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.

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 an ancestor node. Note that the term *inherited attribute* refers to an attribute defined in the ancestor node, and is thus a concept unrelated to the inheritance of object-oriented languages. In this article we will use the term *inherited attribute* in its AG meaning, unless explicitly stated otherwise.

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 ancestor and used for propagating information upwards in the tree, e.g. propagating type information up from an operand to its enclosing expression.

In the JastAdd II implementation of RAGs the declaration of a synthesized attribute can be seen as an abstract method declared in the node holding the attribute. The equations are implemented as overridden methods in the subclasses that originate from productions of the node declaring the attribute.

```
// Declaration of a synthesized attribute
// type of Expr nodes and its default equation
syn Type Expr.type() = TypeSystem.UNKNOWN;
```

```
// Overriding default equation for Add nodes
eq Add.type() = TypeSystem.INT;

// Overriding default equation for Id nodes
eq Id.type() = env().lookup(id()).type();
```

An inherited attributes can be seen as methods where the method declaration and method body may be separated. Inherited attributes have their method body, that defines the behavior, in an ancestral node. An inherited attribute equation defines the behavior for a corresponding declaration of that same attribute in the subtree where the targeted equation node is the root. That way the only dependency on tree structure for that attribute is that the node holding the equation must be an ancestor to the node holding a declaration.

The example below defines an inherited attribute *moduleName* in each *Expr* node. The name of each *ControlModuleType* is propagated down to each expression from an equation in the *ControlModuleType* node that is defined for each subtree rooted at each *member* child.

```
inh String Expr.moduleName();
eq ControlModuleType.member().moduleName() = name();
```

Higher order attributes [VSK89] allow new parts of the tree to be defined as a function of the existing, partially attributed tree. That way, the tree can reflect properties of the tree that are context dependent and can not be built until some semantic analysis has been done. These attributes are higher-order in that the new created trees may contain attributes themselves.

4.3 Rewrite rules

ReRAGs [EH04] extend RAGs with rewrite rules that automatically and transparently rewrite nodes. The rewriting of a node is triggered by the first access to it. Such an access can occur either in an equation in the ancestor 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.

A rewrite step is specified by a rewrite rule that defines the conditions when the rewrite is applicable, as well as the resulting tree. After the application of one rewrite rule, more rewrite rules may 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. Both the rewrite rule application order and the tree traversal order are implicitly defined by attribute dependences.

5 Implementation

This section describes the implementation of the extended control modules language. The abstract syntax as well as the static semantics for the language are modeled using ReRAGs in the JastAdd II tool. An architecural overview of the compiler is first given, and then the base language followed by the extended control modules language. The base language implementation serves as an introduction in how to write a compiler in JastAdd II with modularity and separation of concerns in mind. The implementation has been done with two important design principles in mind. The abstract syntax tree (AST) should be the only data structure, i.e. no external data structures such as symbol tables are used, and all program elements should be explicitly visible in the AST, i.e. all declarations including implicit elements such as primitive type declarations are included in the tree structure.

5.1 Architecture

The lexical analysis and the parsing phase of the compiler have been implemented using traditional tools, e.g. JavaCUP and JFlex. The parser emits an AST that is built according to the abstract grammar specified using JastAdd II. The semantic analysis and code generation of the base language as well as the extended language are implemented in a modular fashion using ReRAGs operating on the tree emitted from the parser. A short overview of the modules in the base langauge is first given, followed by the modules implementing the extended language.

Base language The base language is divided into three modules performing static semantic analysis and code generation. The static semantic analysis of the language is divided in two modules, *Basic name binding* and *Basic type system*. The code generation is then performed by a separate *XML code generation* module.

Basic name binding The basic name binding module is part of the semantic analysis phase. It tries to bind each identifier use-site to its corresponding declaration. An analysis phase then verifies that each identifier has been bound correctly and that there are no multiple defined identifiers.

Basic type system The basic type system module is also part of the semantic analysis phase. It computes the type for each expression and verifies that all operations that work on types have compatible types.

XML code generation This module emits an XML representation of the AST that is compatible with the XML Schema used in the Control Builder tool by ABB Automation Technology Products. The translation from an AST in the base language to the XML representation is trivial and not further discussed in this paper.

Extended language The language extensions are implemented by five separate modules that depend on the semantic analysis modules for the base language. Three of these modules implement semantic analysis for the language extensions: *Extended name binding*, *Extended type system*, and *Protection mechanisms*. The remaining two modules implement the translation from the extended language to the base language: *Inheritance flattening* and *Eliminate references*.

Extended name binding Extends the basic name binding module with scope rules for inherited scopes and a new scope caused by dereferenced references.

Extended type system Extends the basic type system with type hierarchies and extends the type checker to use the subtype relation where appropriate.

Protection mechanisms Adds an additional set of semantic checks that enforce the protection rules.

Inheritance flattening Flattens the inheritance structure by explicitly adding inherited members to type declarations.

Eliminate references Eliminates references by statically computing the target of each reference and replacing all remote reference accesses with direct local accesses.

The following sections presents the design and implementation of each module.

5.2 Basic name binding

This section describes the implementation of name analysis for the base language. The purpose of the name analysis phase is to define the scope rules of

the language and bind each use-site of a name to its corresponding declaration. The name binding implementation for control modules with nested scopes is shown below. Each use-site node has an attribute *decl()* which is a reference to the bound declaration site. That binding mechanism is implemented by letting each node that affects the scope rules define an equation for each use-site within that scope. A parameterized attribute *lookup(String name)* is added to each *Name* and *ControlModuleType*. The equation provided in *ControlModuleType* scans its *Members* for declarations with a matching name and delegates the lookup to the enclosing scope if none is found. The delegation using the *lookup* attribute requires that attribute to be declared in *ControlModulesType* as well and not only in *Name*. Finally the *decl* attribute is added to each *Name* and implemented by calling the *lookup* attribute with the actual name as the parameter. The same technique is used to provide the type scope to lookup type declarations.

```
syn VariableDecl Name.decl() = lookupVar(name());

inh VariableDecl Name.lookupVar(String name);
inh VariableDecl ControlModuleType.lookupVar(String name);

eq ControlModuleType.member().lookupVar(String name) {
  for(int i = 0; i < numMember(); i++)
    if(member(i).isVariableDecl() &&
       member(i).name().equals(name))
      return (VariableDecl)member(i);
  return lookupVar(name);
}
```

The scope rules are not only affected by *ControlModuleType* nodes but also by the common dot notation used to access elements in a record. This is easily implemented by allowing the *Dot* node type change the scope rules for the *lookupVar* attribute as illustrated below. The lookup is delegated to the type declaration for the type of the element on the dot left hand side, i.e. the *RecordType* declaration. If no declaration is found the *unknownVariable()* declaration is returned that enables later stages to assume all names are bound to a declaration.

```
ast Dot : Expr ::= Expr left, Expr right;
eq Dot.right().lookupVar(String name) =
  left().type().lookupVar(name);
syn VariableDecl RecordType.lookupVar(String name) {
  for(int i = 0; i < numElement(); i++)
    if(element(i).isVariableDecl() &&
       element(i).name().equals(name))
      return (VariableDecl)element(i);
```

```
  return unknownVariable();
}
```

To verify that there are no name binding errors, the *decl* attribute in each *Name* is compared to the *unknownVariable()* declaration. As long as there are no *unknownVariable()* bindings, all binding resolve to proper name declarations. There is also a check to find multiple declarations of the same name by verifying that when looking up a name from a declaration that same declaration is returned. If there were multiple declarations with the same name, both lookups would have been bound to the first declaration.

5.3 Basic type system

Type analysis involves the tasks of determining the type of every expression and to check that the types involved in an operation are compatible. An attribute *type* and equations to propagate type information is added to each expression. The figure below shows an example where the *type* attribute as well as the equations for integer literals and binary operations are included.

```
syn TypeDecl Expr.type();
eq IntLiteral.type() = lookupType("String");
eq BinOp.type() = left().type() == right().type() ?
   left.type() : illegalType();
```

The second task is to ensure that when two types are involved in an operation those two types are compatible. This is implemented by adding a type constraint to each node that involves types expressions or declarations. That constraint must be fulfilled for the operation to be correct from a type perspective. For instance, for an assignment statement the type constraint is that the type of the left hand side must be the same as the right hand side, while a function call must ensure that each argument is of the same type as the declared parameter type. An implementation of the described behavior is shown below.

```
syn boolean Assignment.typeError() =
  !source().type() == dest().type();
syn boolean FunctionCall.typeError() {
  for(int i = 0; i < numArgument(); i++)
    if(!argument(i).type() == decl().parameter(i).type())
      return true;
  return false;
}
```

5.4 Extended name binding

When extending the language with inheritance and references, some changes need to be done to the existing name binding implementation. The scope rules are changed since inherited members need to be considered when no matching member is found in the current type. First after all inherited members from the supertypes have been searched the lookup may be delegated to the enclosing scope to allow global variables and nested types. This extension is shown below where a new attribute, *findMember(String name)*, that only looks up members in the current type and possible supertypes, is added to the *ControlModuleType*. The *lookupVar* attribute described in Section 5.2 is changed to use *findMember* and delegate the lookup to the enclosing scope if no proper declaration is found.

```
syn VariableDecl ControlModuleType.findVar(String name) {
  for(int i = 0; i < numMember(); i++)
    if(member(i).isVariableDecl() &&
       member(i).name().equals(name))
      return (VariableDecl)member(i);
  return hasSuperType() ?
    superType().findVar(name) : unknownVariable();
}
eq ControlModuleType.member().lookupVar(String name) {
  VariableDecl decl = findVar(name);
  return decl != unknownVariable() ?
    decl : lookupVar(name);
}
```

The introduction of dereferenced references adds a new scope to the language similar to the dot notation for records. Since dereferencing a reference and element access in a record share the same syntactic notation both operations build identical trees during the context free parsing phase. When the names have been bound and contextual information is taken into account, a rewrite is used to reflect that knowledge in the tree structure. The generic *Dot* node is rewritten to a specialized node type, *RecordAccessDot* or *DereferenceDot*, that reflects the semantic meaning of the operation. This simplifies later stages like type checking and code generation where different equations may be provided for the semantically different operations. The same technique is used on the name nodes to specialize a generic names to variable name or parameter name nodes.

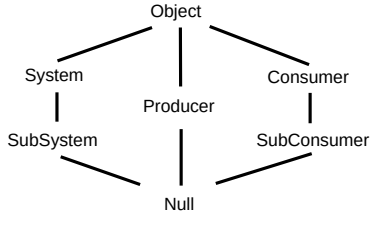
```
rewrite Dot {
  when(left().type().isRecordType())
  to Dot new RecordAccessDot(left(), right());
  when(left(),type().isControlModuleType())
  to Dot new DereferenceDot(left(), right()); }
```

5.5 Extended type system

The extended type system does not affect binding of types but the type attribute and type constraints need to be changed from a strict equality to a subtype relationship. The example in Section 2.2 is changed in that the type constraint for assignment is changed from equality to use the subtype relationship and type propagation for binary operations is modified to propagate the least specific type of the two involved expression or the illegal type if there is not a subtype relationship between the types.

```
eq Assignment.typeError =
  source().type().isSubtypeOf(dest.type());
eq BinOp.type() {
  if(left().type().isSubtypeOf(right().type()))
    return right().type();
  if(right().type().isSubtypeOf(left().type()))
    return left().type();
  return illegalType();
}
```

The subtype relationship is explicitly visible in the source code in that a type declaration that is a direct subtype of another type explicitly inherits that supertype directly through the extends keyword. In the AST the specified supertype is bound to its declaration and the direct subtype relationship can thus be modeled through a reference attribute from the subtype declaration to the declaration of its direct supertype. Primitive types that share a subtype relationship have the same attribute and the same computation is used for both primitive types and user defined types. These reference attributes form an alternative tree within the AST. The tree is turned into a type lattice by adding the null type that is a subtype of every other ControlModuleType. An example type lattice of the control module types that were declared in Section 2.2 and Section 3 is shown below.



The type lattice is used to calculate the subtype relationship by iteratively comparing the closest direct supertype with the target supertype until either the desired type is found or the top type Object is found. The described strategy

can not handle an erroneous declared circular inheritance hierarchy and results in an non-terminating loop. The subtype equation is therefore guarded with a circularity test that ensures that the current TypeDecl is not involved in a circular declaration.

```
syn boolean TypeDecl.isSubtypeOf(TypeDecl decl) {
  if(isCircular()) return illegalType();
  return decl == this ? true : hasSuperType() ?
    superType().isSubtypeOf(decl) : false;
}

syn boolean TypeDecl.isCircular() = testCircular(this);
syn boolean TypeDecl.testCircular(TypeDecl type) =
  hasSuperType() ? type == this :
    superType().testCircular(type);

eq TypeDecl.typeError() = isCircular();
```

5.6 Protection mechanisms

The protection mechanisms does not affect name analysis and type analysis behavior but is merely implemented as a set of additional checks to verify that each access complies to the given protection constraints. An example where each variable access is verified is shown below. The protection level *private* allows only those accesses that originate in the same type as the accessed variable is declared in. To decide in which context an attribute is used or declared, the attribute *hostType* is added, which is a reference to the declaring ControlModuleType for each declaration and variable access. A protection test then uses that attribute to verify that the host type is the same for each variable access as its corresponding variable declaration. The protection level *protected* is implemented in a similar manner but the subtype relation is used instead of strict equality.

```
inh ControlModuleType VariableDecl.hostType();
inh ControlModuleType VariableAccess.hostType();
eq ControlModuleType.member().hostType() = this;

syn boolean VariableAccess.protectionError() {
  VariableDecl decl = decl();
  if(decl.isPrivate() && hostType() != decl.hostType()) {
    return true;
  }
  else if(decl.isProtected() &&
          hostType().isSubtypeOf(decl().hostType())) {
```

```
    return true;
  }
  return false;
}
```

Similar protection constraints are introduced for code blocks that inherit or override existing code blocks in the supertypes. An example where these constraints are enforced for extended code blocks is shown below. The test ensures that there is a code block in the supertype that is not final and that it has an inner statement. A similar technique is used to ensure that all instantiations are valid, i.e. the instantiated type does not contain abstract code blocks.

```
syn String ControlModuleType.hasCodeBlock(String name)
  = lookupCodeBlock(name) != unknownCodeBlock();

syn boolean ExtendedCodeBlock.protectionError() {
  if(!superClass().hasCodeBlock(name())
    return true;
  else {
    if(superClass().codeBlock(name()).isFinal())
      return true;
    if(!superClass().codeBlock(name()).hasInner())
      return true;
  }
  return false;
}
```

5.7 Inheritance flattening

The existing control module run-time system has no support for inherited members from another *ControlModuleType*. Therefore, the inheritance structure is flattened by explicitly adding each member that is inherited from a supertype to the subtype. One may argue that the extended name binding implementation is not needed when all inherited members are added to the subtype. The main reason the implementation is still needed is that the structure is only flattened if there are no compile-time errors, so during error checking the inherited members have thus not been added yet. The rewrite that flattens the class hierarchy is shown below. The rewrite adds all non final members that are not overridden in the subtype as long as there are no static semantic errors. Notice that each added member is duplicated before attached to the *ControlModuleType* since the AST is a tree and not a graph.

```
rewrite ControlModuleType {
  when(noErrors() && !hasAllInheritedMembers())
```

```
  to ControlModuleType {
    for(int i = 0; i < superType().numMembers(); i++) {
      Member m = superType().member(i);
      if(!m.isPrivate() &&
         findMember(m.name()) != unknownMember())
        addMember((Member)m.fullCopy());
    }
    return this;
  }
}

syn boolean ControlModuleType.hasAllInheritedMembers() {
  for(int i = 0; i < superType().numMembers(); i++) {
    Member m = superType().member(i);
    if(!m.isPrivate() &&
       findMember(m.name()) != unknownMember())
       return false;
  }
  return true;
}
```

5.8 Eliminate references

The existing run-time system only allows local variable access except for non local access between parent and child modules through parameters. To be able to generate the base language all remote references must thus be removed. This is done by resolving all dereferenced references and replacing those remote variable accesses by local accesses. All kinds of references, i.e. parameter-, local-, and part object-references, are removed and the primitive instance variables that were accessed remotely by dereferencing a reference are instead accessed through a parameter holding each primitive instance variable. This section describes the technique used to resolve references at compile-time and transform the code to use local accesses instead of remote accesses. To gain local access to any remote instance variable a user defined data type that holds each primitive instance variable is created. A singleton instance of that type, environment, will be created in the root node of the hierarchical system at run-time and then propagated down to all module instances through a parameter. The target module of each reference assignment, used to bind a reference to its target module, can be computed at compile-time since the language is static and due to the single assignment nature of reference assignment.

The following code fragment will be used throughout this section to visualize the various stages in the reference elimination process. A *System* consists of a *Consumer* part object *con*, a *Producer* part object *prod*, and a *Consumer*

remote reference *remote*. A *Producer* has a parameter reference *con* to a *Consumer* and the *Producer prod* in the *System* is initialized to reference the *Consumer* part object *con* in that same *System*. The remote reference *remote* in *System* is initialized by dereferencing two references. The *prod* part object reference in first dereferenced. A remote reference *con* in the first dereferenced reference target module is then dereferenced and the remote reference *remote* is bound to the found target.

```
CONTROLMODULETYPE SYSTEM
  VAR
    con : Consumer (); // Part Object Reference
    prod : Producer (con); // Part Object Reference
    remote : Consumer := prod.con; // Remote Reference
  END_VAR
END_CONTROLMODULETYPE

CONTROLMODULETYPE PRODUCER
  VAR_IN
    con : CONSUMER; // Reference
  END_VAR_IN
END_CONTROLMODULETYPE

CONTROLMODULETYPE CONSUMER
END_CONTROLMODULETYPE
```

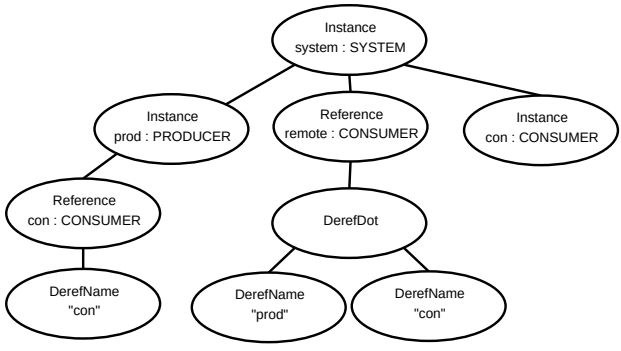
Instance composition structure To evaluate the target module, a hierarchical Control Module instance structure is built that represents the static composition of Control Module instances in the actual program instance. The instance structure contains all instantiated ControlModuleTypes as well as all reference operations, i.e. declarations, assignments, and dereferences. The abstract grammar shown below models the instance tree. There are two different kinds of declarations, References and Instances. References are remote binding to other References and Instances. The references are initialized with a single Reference assignment that is dereferencing other references. An Instance is an instantiated ControlModuleType that has a set of children that are either references or nested ControlModule instances. Reference assignment contains dereferences that are modeled through Dereferences node holding simple and complex names. A simple name is a single dereferenced reference while a complex name is constructed from multiple dereferences.

```
ast Decl ::= <String name>;
ast Reference : Decl ::= Dereference deref;
```

```
ast Instance : Decl ::= <String typeName>, Decl decls*;

ast abstract Dereference;
ast DerefName : Dereference ::= <String name>;
ast DerefDot : Dereference ::= Name left, Name right;
```

The figure below shows the instance structure that corresponds to the *System* code fragment. The subtree is built dynamically during compile-time and attached to the main tree as an higher-order attribute in the root node.



The environment data type holds all instance variables that can be found in the instance structure of the actual program. Each instance variable receives a unique name by concatenating the names of the part object references that are describing the path from the root to the instance variable with the variable name itself. The example below implements this concatenation by adding the name of each *Instance* node through an inherited attribute. The instances in the above example will thus be named *system*, *systemprod*, *systemcon*.

```
syn String Instance.fullName() = parentName() + name();
inh String Instance.parentName();
eq Instance.instance().fullName() = fullName() + name();
```

Reference binding To resolve the reference in the instance structure each dereferenced reference needs to be bound to its corresponding declaration. An example implementation modeled in a very similar fashion as the name binding described in Section 5.4 is shown below. Each instance has a parameterized lookup attribute that searches the declarations for a declaration matching the name parameter. At this level there are no nested scopes and delegation is not needed and an illegalDecl is returned directly if the declaration is not found. The declaration is propagated as an inherited attribute to child instances that

need to lookup the references used in the right hand side of reference assignments. The same attribute is used when complex names are used to directly access a name used in the right child of a DerefDot node.

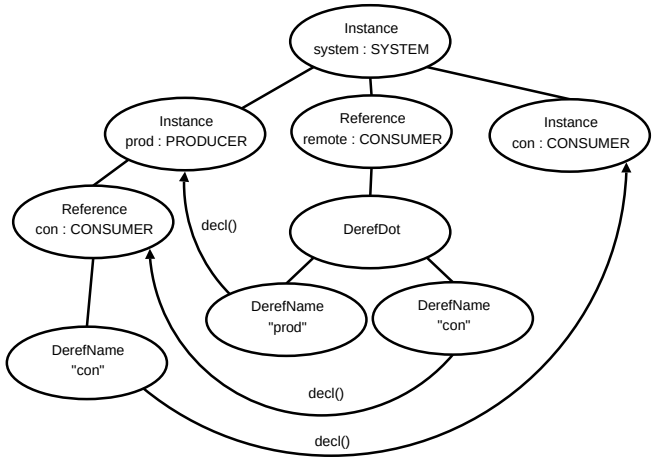
```
syn Decl Instance.instance().lookup(String name) {
  for(int i = 0; i < numDecls(); i++)
    if(decls(i).name().equals(name()))
      return decls(i);
  return illegalDecl();
}

inh Decl Reference.lookup(String name);
eq Instance.instance().lookup(String name) =
  lookup(name);

eq DerefDot.right().lookup(String name) =
  left().decl().lookup(name);

syn Decl Dereference.decl();
eq DerefName.decl() = lookup(name());
eq DerefDot.decl() = right().decl();
```

The figure below show the instance structure after each name has been bound the its corresponding declaration. Notice how the changed scope rules in the *DerefDot* node has affected the binding of the right *DerefName con*.

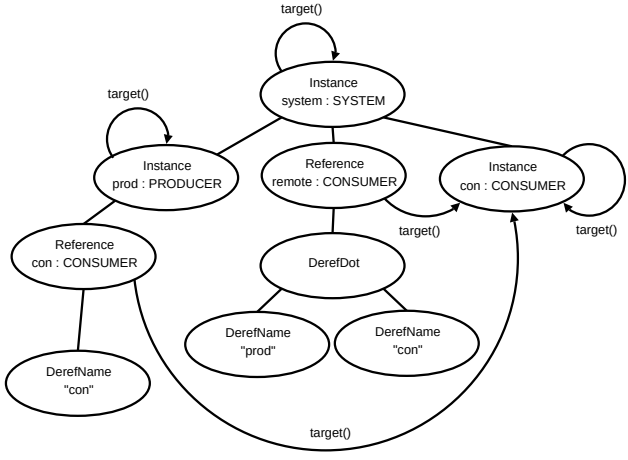


Compute reference target When all names have been bound to a corresponding declaration site the target Control Module instance can be computed. The

target for an Instance node is trivial since it is the Instance node itself, while References need to find the target by computing the target of the dereferenced references in the right hand side of each reference assignment. The attributes that implement this final stage in resolving references is shown below. The final stage uses these attributes to rewrite the code locks to use these attributes effectively replacing each dereferenced reference with an access to the corresponding element in the environment structure.

```
syn Instance Decl.target();
eq Reference.target() = deref().decl().target();
eq Instance.target() = this;
```

The final figure shows how the *target()* attribute binds each remote reference initialization to an actual Control Module instance.



The previous examples are somewhat simplified since they do not handle reference circularities, which are illegal, and local references. Circularity in reference assignments can be handled by adding a test similar to the one used for the *isSubtypeOf* attribute in Section 5.5 to break circularity and generate a compile-time error. Local references are not included because of their similarity to parameter references. They only differ in the scope rules used when binding references in the right hand side in reference assignments. Parameter references look up references in the parent of the instance declaring the parameter while local references look up references in the declaring instance itself.

Replace reference with local access During code generation a new Control Module type in the base language is created for each instantiated module in

the instance structure. These types are uniquely named using a similar strategy as for the local instance variables. Each created type instantiates the part objects derived from the original model and initializes the parameters for each part object. All remote accesses to instance variables through dereferencing a reference are turned into parameters that are named using the same concatenation technique as for local variables. There will thus be a parameter for each instance variable that is accessed through remote access. Each reference has been resolved and bound to a specific module instance through the target attribute and the parameters can thus easily be initialized to access the correct target element in the environment instance. Initialization of the environment instance parameter is a simple argument copy and the parameters of primitive types from the original model are left unchanged. Local instance variables are transformed into parameters and initialized to use the target instance in the environment data type. Since this is all done at compile-time the remote references are completely removed during code generation.

6 Conclusions and future work

The Control Module language has been extended with object-oriented features improving encapsulation, code re-use, and static type safety. Encapsulation is improved in that implementation details, such as instance variables, can be kept local in a control module instead of the more global approach needed when data must be placed in a common ancestor module. Code is effectively re-used through inheritance of both behavior and composition. The object-oriented type system improves safety in that a larger set of semantic errors can be found at compile-time.

The base language as well as the language extensions were modeled using ReRAGs in the JastAdd II tool. Using the AST as the sole data structure was successful but relied heavily on dynamic tree construction through rewrites and higher-order attributes. The dynamic restructuring of the AST also simplified computations in that context-sensitive semantic properties of the AST could be made explicit in the tree structure. The various phases in semantic analysis, e.g. name binding and type checking, as well as the language extensions, e.g. inheritance and protection, could be implemented in separate modules providing good separation of concerns.

All the extended langauge features are translated into base language constructs and the AST for the extended language is thus transformed into an AST in the base language. That tree is exported to the ControlModule development environment and can thus benefit from existing infrastructure such as native compilers and run-time system.

There are several interesting topics for future work:

- We have implemented a full static semantic analyzer for Java 1.4 using a similar strategy as the one described in this paper and are currently extending the implementation with Java 1.5 features like generics, improved for-stmt, static imports, etc. Our current results indicate the we are able to create modular extensions for the new Java 1.5 language features using techniques similar to the ones described in this paper and generate Java 1.4 code through tree transformations.
- By transforming the extended language to the base language we can re-use the existing infrastructure such as compilers and run-time system. There is, however, no support for debugging code in the extended source code language. It would be interesting to provide the debugger with a mapping from the extended language to the base language to enable source level debugging in the extended language as well. This could probably be done in a manner similar to the "JSR-45 - Debugging support for other languages" proposal that supports source level debugging for other languages than Java as long as they generate Java virtual machine byte-code.

References

[DMN68] O. J. Dahl, B. Myrhaug, and K. Nygaard. *SIMULA 67 Common Base Language*. Norwegian Computer Center, Oslo, 1968.

[EH04] Torbjörn Ekman and Görel Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004: 18th European Conference on Object-Oriented Programming*, 2004.

[Ekm04] Torbjörn Ekman. A case study of Separation of Concerns in Compiler Construction using JastAdd II. In *Proceedings of the Third AOSD workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, 2004.

[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.

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

[JT01] Karl-Heinz John and Michael Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems. Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Tool*. Springer-Verlag, 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).

[MMPN93] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-oriented Programming in the BETA language*. Addison-Wesley, 1993.

[TC79] B. Taconet and B. Challot. Grafcet programming on programmable logic controller with logical, ladder and boolean language. *Revue Nouvel Automatisme*, 24(1-2), 1979.

[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.