Object-Oriented Declarative Program Analysis

Eva Magnusson

Doctoral Dissertation, 2007

Department of Computer Science
Lund University

ISBN 978-91-628-7306-6
ISSN 1404-1219
Dissertation 28, 2007
LU-CS-DISS:2007-2

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

Email: Eva.Magnusson@cs.lth.se
WWW: http://www.cs.lth.se/home/Eva_Magnusson

Typeset using TeXShop
Printed in Sweden by Tryckeriet i E-huset, Lund, 2007

ABSTRACT

This thesis deals with techniques for raising the programming level for a particular kind of computations, namely those on abstract syntax trees. Such computations are central in many program analysis tools, such as compilers, smart language-sensitive editors, and static analysis tools. All techniques presented in this thesis support modular description and efficiency and are capable of handling large programs.

The work is based on Reference Attributed Grammars (RAGs) which combines object-oriented features with declarative programming to specify computations on abstract syntax trees. RAGs have proven useful, e.g., for performing static-semantic analysis of object-oriented languages. We investigate new applications of RAGs, extensions of RAGs in order to cover yet more applications, modularization issues for RAGs, and implementation of RAG extensions.

The thesis consists of an introduction and four papers. The first paper deals with the application of RAGs to a new problem area: program visualization. The second paper describes JastAdd, a practical system for RAGs, based on static aspect-oriented programming and which supports the combination of imperative Java programming with declarative RAG programming. JastAdd has been used for developing practical compilers for full-scale languages. The third paper describes CRAGs, an extension of RAGs allowing circular dependencies and where the evaluator computes fixed-point solutions by iteration. CRAGs open up RAGs for new application areas such as grammar and data flow analyses. The fourth paper deals with extending attribute grammars with collection attributes and circular collection attributes. These attributes allow whole-program properties such as cross-references to be easily specified. A number of evaluation techniques and a number of application areas, including source code metrics, are described. All techniques described in the papers have been implemented and tested in practice.

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 am particularly grateful to my supervisor Görel Hedin who introduced me to the field of Attribute Grammars. Most of the work presented here has been carried out jointly with her. She has always supported me with good advice and lots of encouragement and feedback. My postgraduate studies have been extended over a long period since simultaneously I worked as a teacher and director of undergraduate studies at our department. It has been of great importance to me that Görel always encouraged me to continue and made me believe that one day I would finally finish my thesis.

I would also like to thank Torbjörn Ekman who has been active in several projects involving the JastAdd tool. Torbjörns help and support has been invaluable, especially when working with collection attributes as described in the last paper of this thesis. I am deeply indebted to him for accommodating me with so much time.

Other members of the research group as well as other members of the department staff have also contributed in different ways. Professor Boris Magnusson, the leader of the research group, encouraged me when, after many years of teaching, I expressed an interest in taking up graduate studies again. Sven Gestegård Robertz let me use a language developed in his Masters thesis as an example of how to integrate visualization in state transition languages. Mathias Haage provided some good tips on various work on visualization. Anders Ive provided constructive comments especially on the second paper of this thesis. Christian Andersson, Christian Söderberg and Per Holm helped me to prepare final versions of papers for publication by sharing their expertise in Latex. Per Anderson and Anders Nilsson generously shared their experience in using Latex when I prepared this thesis. Peter Möller, Anne-Marie Westerberg and Lena Ohlsson have helped me with various practical things. Many thanks to all of you.

Finally, but not least of all, I would like to thank my husband Jan. Over they years he always encouraged me to continue my graduate studies. He also helped me in so many other ways. Most importantly, he helped me by always being there for me.

This work has been performed within the Center for Applied Software Research (LUCAS) at Lund Institute of Technology. It has been funded partially by the Swedish Agency for Innovation Systems (Vinnova) and the Swedish Research Council (Vetenskapsrådet).

PREFACE

This thesis consists of an introductory part and four papers. The research papers included in the thesis are

E. Magnusson, G. Hedin. Program Visualization using Reference Attributed Grammars, Nordic Journal of Computing 7 (2000) 67-86. ©Publishing Association Nordic Journal of Computing, Helsinki. An earlier version of this paper was presented at NWPER'00, Nordic Workshop on Programing Environment Research. Lillehammer, Norway. May 2000.

G. Hedin, E. Magnusson. The JastAdd system – an aspect-oriented compiler construction system, SCP - Science of Computer Programming, 47(1):37-58. Elsevier. November 2002. ©Elsevier. An earlier version of this paper was presented at LDTA'01, First Workshop on Language Descriptions, Tools and Applications, Genova, Italy. April 2001.

E. Magnusson, G. Hedin. Circular Reference Attributed Grammars – their Evaluation and Applications, SCP - Science of Computer Programming, 68(1):21-37. Elsevier. August 2007. ©Elsevier. An earlier version of this paper was presented at LDTA'03, Third Workshop on Language Descriptions, Tools and Applications, Warsaw, Poland. April 2003.

E. Magnusson, T. Ekman and G. Hedin. Collection Attribute Algorithms. Manuscript September 2007. An extended version of: E. Magnusson, T. Ekman and G. Hedin. Extending Attribute Grammars with Collection Attributes – Evaluation and Applications accepted at SCAM 2007, The Seventh IEEE Working Conference on Source Code Analysis and Manipulation, Paris September 2007. Best paper award, SCAM 2007.

There is also a technical report with a more detailed description of how program visualization presented in the first paper has been implemented. This report is not included in the thesis

> E. Magnusson. State Diagram Generation using Reference Attributed Grammars. Technical report LU-CS-TR:200-219, Department of Computer Science, Lund University, Lund. March 2000.

CONTENTS

CHAPTER I

INTRODUCTION

An important goal in computer science is to raise the level of programming, providing languages that are closer to the way the programmer thinks and reasons. The development of object-oriented programming languages is one important step in this direction, providing abstraction mechanisms for real-world modelling containing both data and computation aspects. Declarative programming is another important principle, allowing the programmer to describe what to be computed without having to explicitly state in which order the computations should take place. Modularization is a third important principle, allowing reuse and separations of concerns.

This thesis deals with techniques for raising the programming level for a particular kind of computations, namely those on abstract syntax trees (ASTs). Such computations are central in many program analysis tools, such as compilers, smart language-sensitive editors, and static analysis tools. All techniques presented in this thesis support modular description and efficiency and are capable of handling large programs.

Our basis is Reference Attributed Grammars (RAGs) [15], which combines object-oriented features with declarative programming to specify computations on abstract syntax trees. RAGs have proven useful, e.g., for performing static-semantic analysis of object-oriented languages. A RAG is an extension of traditional or classical attribute grammars (AGs) [23] which is a formalism in which the static semantics of a programming language can be specified using a declarative approach. Traditional AGs are often considered clumsy and difficult to use for tasks where computations on general graphs are natural. Examples include name- and type-analysis of languages with complex scope rules, and many other static analysis problems. By allowing references between distant nodes in the AST, the RAG formalism supports the definition of graphs as attributes, facilitating these tasks. Furthermore, the object-oriented view of the grammar used in RAGs is a conceptual extension of AGs which makes it possible to apply all the object-oriented modularization advantages such as inheritance and method overriding.

In the thesis we deal with extensions of RAGs and how they can be combined to facilitate tasks further for the attribute grammar author. We also explore applications of the combined extensions. Furthermore, we discuss evaluation techniques for extended AG formalisms. Some extensions concern the formalism, e.g., allowing circular dependencies between attribute instances or introducing collection attributes, while others can be characterized as conceptual. Conceptual extensions include the object-oriented modelling of the grammar and modularization concepts.

Even in extended AG formalisms, some computations can be complicated or unnatural to express. Another objective of this thesis is therefore to show how the declarative approach used in attribute grammars can be combined with imperative techniques in a tool using static aspect-oriented modularization and where modules can be implemented in either imperative or declarative style.

The thesis consists of an introduction and four papers. The objectives of the research presented in the papers can be summarized in the following way:

- *Application areas* for RAGs and their extensions, especially outside the traditional compiler-related area.
- *Modularization* of attribute grammars as a way to separate subtasks according to aspect, to support reuse and to open up the possibility to combine imperative and declarative style modules.
- *Extensions of RAGs*, how they widen the applicability of RAGs and facilitate for the grammar author.
- *Evaluator implementation* for the extended formalisms.

The first paper deals with the application of RAGs to a new problem area; program visualization. The second paper describes JastAdd, a practical system for RAGs, based on static aspect-oriented programming and the combination of imperative and declarative programming. The third paper describes an extension of RAGs supporting circular dependencies (CRAGs). The extended formalism opens up for new application areas such as grammar flow and data flow analysis, which is also discussed in the paper. The fourth paper describes how RAGs can be extended with the collection attribute mechanism, allowing declarative specifications of whole program properties such as cross references. A number of evaluation algorithms are presented as well as application areas, including source code metrics. The combined formalism, circular collection attributes, is also introduced.

The techniques have all been implemented and tested in practice. Our tool JastAdd incorporates the extended AG formalism and automatically generates an evaluator for it. Its modularization concept also allows imperative style modules to be freely combined with AG modules. Thanks to the work of Torbjörn Ekman on other parts of JastAdd, as well as a full Java implementation using JastAdd, it

has been possible to try out example applications computing properties of large Java programs.

The aim of this chapter is to describe and motivate the objectives further and to present the contributions. In order to do so, an overview of the attribute grammar formalisms and their corresponding evaluation techniques is given first. Then it is possible to present the objectives of our research in a more detailed manner and to give short descriptions of each paper. Finally, the thesis contributions are summarized and possible directions for future work are discussed.

The rest of this chapter is organized as follows: Section 1 introduces traditional attribute grammars and some extended formalisms. Section 2 is devoted to evaluation techniques. In Section 3 the objectives of this thesis work are discussed and the papers are presented. Section 4 concludes the introduction and discusses some possible future work.

1 Attribute Grammars

1.1 Traditional Attribute Grammars

Traditional attribute grammars (AGs) were introduced by Knuth [23] in 1968 as a formalism to specify the syntax and semantics of a programming language. An AG is a context-free grammar, a set of *attributes* associated with its nonterminals, and a set of *equations* specifying the values of the attributes.

The nodes of an abstract syntax tree (AST) are instances of nonterminals. The attributes $A(X)$ of a nonterminal X consists of two subsets: the *synthesized attributes* and the *inherited attributes*. Each attribute is specified by an equation. Synthesized attributes propagate information upwards in the abstract syntax tree and inherited attributes propagate information downwards. For example, a synthesized attribute `type` of a nonterminal `Identifier` can be used to propagate information upwards to check for correct use of identifiers in expressions. An inherited attribute `env` containing information about declarations can be used to propagate information downwards and be used to look up information about the type of an identifier.

AGs use a declarative formalism, i.e., it specifies what to do but without imposing any explicit order of computations. For problems where declarative specifications are suitable, the specifications are often clear and concise and easy to extend. They also often help avoiding errors common in imperative style programming.

Consider a production $X_0 ::= X_1X_2...X_k$ of a context-free grammar. An equation specifying the value of an attribute a_0 is written $a_0 = f(a_1, a_2, ...a_n)$. The equation defines the value of a_0 in terms of its semantic function f. Each equation only involves information associated with the local symbols of the production, i.e., the attributes $a_1, ..., a_n$ must be attributes associated with the symbols, $X_0, X_1, ...X_k$. An equation defines either a synthesized attribute of the nonterminal of the left-hand side, X_0, or an inherited attribute of one of the right-

hand side nonterminals $X_j, 1 \leq j \leq k$. The arguments of the semantic function, $a_1, a_2, ...a_n$, can be attributes of any of the symbols $X_0, X_1, ...X_k$.

In order for the AG to be *well formed* there must be exactly one equation defining each attribute of any syntax tree. This requirement is fulfilled if for each production there is one equation for each synthesized attribute of X_0 and one for each inherited attribute of all the right-hand-side symbols $X_j, 1 \leq j \leq k$. The start symbol of the grammar must not have any inherited attributes.

A semantic function introduces dependencies between attributes. If a_1 is used to define a_0, then a_0 is dependent on a_1. Traditional AGs consider circular dependencies as an error, i.e., for any syntax tree derivable from the grammar there must not be any circular dependencies between instances of attributes.

Attribute *evaluation* means assigning values to attribute instances. An attribute instance is said to be *consistent* if its value is equal to the application of its semantic function, or, in other words, if its equation is satisfied. An attributed abstract syntax tree is consistent if all its attribute instances are consistent, i.e., the equations of all its attribute instances are satisfied. The values of the attribute instances of a consistent AST is a solution to the equational system made up of all the equations of its attribute instances. An attribute grammar is said to be *well defined* if every possible AST has exactly one solution.

An *attribute evaluation scheme* is a method for obtaining consistent attribute assignments. Some schemes evaluate all attribute instances and obtain consistently attributed ASTs. Other schemes evaluate individual attributes on demand by evaluating only the subset of attribute instances of the AST of which the demanded attribute is dependent. We will describe some evaluation techniques further in Section 2.

1.2 Reference Attributed Grammars

Traditional AGs provide a concise way of specifying local dependencies. However, many tasks require information to be transmitted between distant nodes in the abstract syntax tree. One example is name analysis of programming languages, especially those with advanced scope rules. To specify name analysis in the traditional AG formalism you need to replicate large complex aggregate attributes containing the necessary declaration information to all nodes representing use sites. For languages with complex scope rules, like object-oriented languages, the task becomes very cumbersome.

Several researchers, e.g., [6, 15, 27], have suggested extensions to AGs by allowing attributes to be references to remote nodes in the syntax tree and to use those references to access attributes of the remote nodes. When specifying name analysis you may then link each use site directly to its corresponding declaration site by a reference attribute. Information needed for example in type analysis can then be accessed from use sites. The syntax tree itself is in this way used as a

symbol table and there is no longer any need to replicate information all around the tree.

This thesis work is based on Reference Attributed Grammars (RAGs) as suggested by Hedin [15]. This extension supports attribute access via references. Attributes are allowed to reference nodes in the abstract syntax tree and collection-valued attributes may contain reference values. A reference attribute may be dereferenced to access attributes in the remote node. RAGs facilitate specification of problems where non-local dependencies are common and they have been used in problem areas such as program visualization (described in the first paper of this thesis), specification of object-oriented languages [15], design pattern checking [8], and prediction of worst case execution times [26].

RAGs were initially implemented in a tool APPLAB [5], developed at our department. APPLAB is an interactive environment based on language-sensitive editing, aimed at the interactive design of domain-specific languages. APPLAB was used for the implementation of the work described in the first paper of this thesis. The second paper describes a new tool, JastAdd, which is a compiler construction tool supporting RAGs, static aspect-oriented modularization, as well as the combination of imperative style and declarative style modules. The JastAdd system, described in the second paper, was later re-implememented by Torbjörn Ekman [9]. The new version was bootstrapped in itself and extended to allow rewriting of the AST [11]. It has also extended support for parameterized attributes and various short-hands. The current version of JastAdd [1] has later been enhanced to handle circular grammars as well as collection attributes as described in the third and fourth papers.

1.3 Attribute Grammars with Collection Attributes

In RAGs, reference attributes are used to read information from referenced sites in the AST. Allowing information to flow in the opposite direction in more or less restricted forms has been suggested by a number of researchers. One restricted form was suggested by Knuth [23] who allowed information to be propagated in this way to global sets in the start symbol. Other researchers suggested the introduction of *collection attributes*, whose values are defined as a combination of properties in distant AST nodes. Kaiser [20] and Beshers [4] allowed collection attributes associated with subtrees. Hedin [14] introduced general collection attributes but with partly manual implementations techniques.

In our own work we implemented an extension of AGs with collection attributes based on the work of Boyland [6]. Here, a collection attribute is defined through a number of partial definitions, located in arbitrary AST nodes and are typically used for cross-referencing information. The final value of a collection attribute is defined as the application of a combination operation to its partial definitions. For each partial definition, a reference attribute is used to point out for which instance of the collection attribute the definition is intended. In the fourth

paper we introduce a number of implementation algorithms for this extension. These algorithms have all been implemented in the JastAdd tool.

1.4 Circular Attribute Grammars

Many computations on abstract syntax trees are easily specified using recursive equations, often broadly distributed over the tree and introducing circular dependencies. Examples come from different problem areas like data-flow analysis and live analysis in optimizing compilers, and properties of circuits in hierarchical VLSI design systems [12, 17].

In a traditional AG it is considered an error if circular dependencies between attribute instances occur for any derivable syntax tree. However, as several researchers have pointed out, attribute grammars with circular dependencies can under certain constraints be considered well defined in the sense that all equations can be satisfied [12, 17].

The most common way to ensure that circular AGs are well defined is to require that the domain of attributes involved in circular definitions can be arranged in lattices of finite height and that the semantic functions defining the attributes involved are monotonic with respect to these lattices. If these conditions are fulfilled, a least fixed point can be calculated using an iterative process as in Fig. 1.

```
for each attribute x_i involved in the cycle
  x_i = ...// initialize each attribute in cycle to a bottom value;
repeat {
  for each attribute x_i in the cycle
    x_i = f_i(...)
} until (no computation changes the value of an attribute)
```

Figure 1: Iterative algorithm for computing the least fixed point for attributes on a cycle. f_i denotes the semantic function of the attribute x_i.

Allowing circular dependencies under proper constraints makes many specifications easy to write for the AG author and easy to read and understand. The specifications involving circular dependencies are often a direct translation of their mathematical recursive definitions. Farrow [12] uses as an example the specification of a language where the use of a constant is allowed before its declaration. He shows how its alternative noncircular specification, in contrast, adds huge complexity using, e.g., higher order functions one of which in essence captures the iterative process used in Fig. 1.

The third paper of this thesis describes the possibility and advantages of combining circular attribute grammars with reference attributed grammars. In the fourth paper the combination of circular grammars and collection attributes is introduced.

1.5 Object-Oriented Attribute Grammars and Modularization

From an object-oriented perspective the nodes of a syntax tree can be viewed as objects of classes corresponding to the productions of a grammar. These objects have children corresponding to production right-hand sides [13]. Each nonterminal X can be modelled as an abstract class and its different alternative productions can be modelled as concrete subclasses. Attribute declarations are in traditional AGs associated with nonterminals and their defining equations with productions. A synthesized attribute `a` of X can in the object-oriented perspective be modelled as a virtual function `a()` and a semantic function of a production defining `a` is modelled as an implementation of the function `a()`.

From the object-oriented perspective it is also desirable to allow equations to be associated with nonterminals. They then model the default behavior, which may be overridden by equations in some of their subclasses. It is also convenient to allow the introduction of abstract superclasses that do not correspond to any of the nonterminals of the grammar. For example, it might be convenient to introduce an abstract class `ASTNode` and make this the root class of the class hierarchy. Behavior common to all node classes can then be modelled by attributes of the class `ASTNode`.

Object-oriented AGs (OOAGs) is not an extension of traditional AGs – any OOAG can trivially be reformulated as a traditional AG. Rather, the difference is conceptual. By formulating the underlying context-free grammar as a class model, the syntax tree can be directly understood as a tree of objects, and all the object-oriented advantages of inheritance and overriding can be applied, yielding concise specifications that are easy to understand and write by people with an object-oriented programming background. Furthermore, the object-oriented model of AGs allows easy integration with imperative object-oriented program code, as is discussed further in the second paper.

In object-oriented programming, class hierarchies are used for modularization purposes. For many tasks, for example compiler construction, this type of modularization is not sufficient. Each node class will contain code related to several different subtasks such as name analysis, type checking, code generation etc. Attribute grammar systems normally introduce another type of separation mechanism by allowing specifications to be textually split into modules. The AG author may then specify appropriate attributes and their equations in different modules, e.g., according to different aspects of the actual problem. The union of all modules constitute the attribute grammar. The combination of object-oriented attribute grammars with a mechanism supporting such aspect-oriented modularization has many advantages. Features from object-orientation, like inheritance and overriding, make many specifications easier to express and a modularization mechanism supports reuse, modification and extension of existing modules in different applications. These issues are further addressed in the second paper.

2 Evaluation Techniques for Attribute Grammars

As mentioned earlier, evaluating an attribute instance means assigning it a value so that its equation is satisfied. In the following subsections we will describe some general techniques for traditional AGs and how they can be adapted to handle extended AG formalisms.

2.1 Evaluation of Traditional Attribute Grammars

Evaluators for traditional attribute grammars can be constructed automatically by several techniques. These techniques all have in common that they evaluate attributes in an order based on the attribute dependencies: an attribute is not evaluated until all the attributes it depends on have been evaluated. This allows *optimal* evaluation, meaning that each attribute is evaluated at most once.

Many techniques construct dependency graphs and use these to compute the evaluation order. An evaluation technique is usually classified as static or dynamic depending on when the dependency graph is constructed. For *static* techniques, the dependency graphs are constructed at evaluator construction time, based on the attribute grammar. For *dynamic* techniques, the dependency graphs are constructed at evaluation time, based on the AST. The static dependency graphs are pessimistic approximations of all possible dynamic dependency graphs, and are therefore usually less general than dynamic techniques, but usually they yield faster evaluators.

Another way of characterizing an evaluation technique is if it is *data-driven* or *demand-driven*. A data-driven evaluation technique uses the dependency graph to evaluate all attribute instances in an order corresponding to the topological sort of the graph, and stores them in memory cells. A demand-driven technique uses no explicit dependency graph. Instead, it makes use of the fact that the graph is implicitly defined by the semantic functions. Accessing an attribute is realized by calling its semantic function and only attributes needed for computations of demanded attribute instances are computed.

More detailed descriptions of the different groups of evaluation techniques for traditional AGs are given in the following subsections.

Static data-driven techniques

Static data-driven techniques evaluate all attribute instances. They use the grammar to derive information about possible dependencies between attribute instances in order to construct a proper evaluation scheme. Evaluators constructed with this technique therefore do not perform any run time analysis. An example of a static method designed to handle noncircular AGs is the one proposed independently by Katayama [22] and Courceller & Franchi-Zannettacci [3]. In their scheme all possible dependencies between attributes of each production are derived from the grammar. Using these graphs, a set of mutually recursive functions to evaluate the

attributes are constructed. The technique is not completely general in that it sometimes fails to build an evaluator even if the AG is noncircular. This is because the derived dependency graphs may contain circularities introduced by spurious edges that could not appear in any syntax tree.

There are also a number of subclasses of traditional AGs for which there exists static construction techniques that produce especially simple and fast evaluators, while being sufficiently general to handle full programming languages, e.g., Ordered Attribute Grammars [21].

Dynamic data-driven techniques

Dynamic data-driven techniques analyze dependencies between attribute instances at evaluation time for the abstract syntax tree at hand. All attribute instances are evaluated.

An example of a general dynamic technique is the one proposed by Jones [17]. The dependency graph is derived at evaluation time and is used as a basis for evaluating the attributes in proper order. If the AG is noncircular, the dependency graphs for all possible ASTs are acyclic. Attributes can therefore be evaluated by simply applying their respective semantic functions according to the topological ordering of the dependency graph. The scheme is optimal in the sense that every attribute instance is evaluated only once.

Demand-driven techniques

Demand-driven techniques evaluate only attributes needed for computing demanded attribute instances. For demand-driven evaluation there is a simple and general evaluation technique which replaces each attribute by its semantic function. Accessing an attribute is realized by calling its semantic function. This evaluation technique does not construct any explicit dependency graphs, but makes use of the fact that the semantic functions define the dependency graph implicitly. This technique was described in [16, 18, 24].

The plain demand-driven technique can be non-optimal since the same semantic function might be called many times. In the worst case, the time complexity is exponential in the number of attributes. To overcome this problem, attributes may be cached when they are evaluated for the first time. When an attribute is demanded for evaluation it is checked if it has already been computed. In that case its cached value is returned. Otherwise its semantic function is called, the resulting value is cached, and a flag is set to mark the attribute as computed. If all attributes are cached, optimal evaluation is achieved at the cost of memory space. An alternative is to let the AG author decide which attributes to cache.

Since a demand-driven technique does not necessarily evaluate all attribute instances, its performance can be better than a data-driven technique.

The object-oriented view on attribute grammars presented in Section 1.5 makes it very easy to automatically generate demand-driven evaluators. In fact, the eval-

uator is constructed implicitly by translating the attributes and their equations to virtual functions and their implementations. This is discussed in more detail in the second paper.

2.2 Evaluation of Reference Attributed Grammars

The static technique of Katayama and the dynamic technique of Jones described earlier cannot be applied to RAGs. In a RAG the dependency graph is not known completely before evaluation. Dependencies are introduced by reference attributes and their values will not be known until they have been evaluated.

The demand-driven technique described in Section 2.1 requires no initial dependency analysis, and is immediately applicable to RAGs. It automatically traverses the dependency graph depth-first, evaluating the attributes in topological order. This evaluation technique is used both by the APPLAB tool [5] and by our JastAdd tool described in paper 2, as well as by other systems for similar formalisms, e.g., [6, 27] .

2.3 Evaluation of Circular Attribute Grammars

Farrow [12] showed that the static technique of Katayama can be generalized to handle circularities. The possibly circularly defined attributes are detected by identifying the strongly connected components of the production dependency graphs. Components consisting of one attribute instance only are treated as in the original algorithm. A component with more than one vertex corresponds to attribute instances that are all dependent on each other and they are evaluated together by an iterative process as described in Fig. 1.

Jones [17] showed that his dynamic technique can also be adapted to handle circular AGs. The strongly connected components of the dependency graph are identified. A new graph is constructed by contracting each component into a single vertex. The new graph is acyclic and can be ordered topologically. A vertex corresponding to a single vertex in the original graph is evaluated by applying its semantic function. Attribute instances of a vertex corresponding to more than one vertex in the original graph are evaluated together by a fixed-point iteration.

In the third paper we show that it is possible to adapt the demand-driven technique to handle also circular attribute grammars. Since the demand-driven technique is also applicable to reference attributed grammars it thereby becomes a technique that is capable of evaluating grammars of the combined extended formalism, circular reference attributed grammars (CRAGs).

In the fourth paper we show how circular collection attributes can be evaluated using techniques built on combinations of the evaluation technique for CRAGs and techniques for evaluating non-circular collection attributes.

2.4 Evaluation of collection attributes

The value of a collection attribute is defined as the combinations of partial definitions from remote nodes in the AST. The collection attribute mechanism thus relies on RAGs since reference attributes must be used to point out the nodes in the AST for which the partial definitions are intended.

In the fourth paper we show that the demand-driven technique can be used to evaluate collection attributes. This is potentially very expensive as the complete AST has to be traversed to find contributors for a particular collection attribute instance. For efficiency reasons it is therefore preferable to use techniques which are not purely demand-driven in the sense that more attribute instances than are actually demanded may be partially evaluated. For example, all contributors to all instances can be found during a single tree traversal. When an instance is demanded, a tree traversal can be performed caching information about contributors for each instance of the collection attribute. Subsequent demands for other instances can then use the cached information which speeds up their evaluation. This two-phase technique, as well as a one-phase technique which deviates further from pure demand evaluation, is described and analyzed in the fourth paper.

3 Objectives of this Work and Description of the Papers

The research presented in this thesis has the following objectives:

Applications The traditional application area for AGs is related to compiler construction. One aim of this thesis has been to explore new applications outside the traditional ones and how different extensions of attribute grammars open up new areas of applications. This aspect of our work is addressed in papers 1, 3 and 4.

The first paper focuses on a new application for RAGs; program visualization. In the third and fourth paper we show how extending the formalism to allow circular dependencies and collection attributes further strengthens the expressiveness and applicability.

Modularization By modularization we here mean the textual separation of different parts of the attribute specification. Modularization of the underlying context-free grammar is also an important topic, but it is not addressed in this thesis.

RAGs introduces an object-oriented view of the grammar, modelling nonterminals as abstract classes and productions as concrete subclasses. In object-oriented programming, class hierarchies are used for modularization purposes. Classes will often contain code related to several different subtasks.

An additional modularization concept allowing the code of a class to be textually split over several modules is therefore desirable.

In RAGs the object-oriented view is used to model the abstract syntax tree. The computations within each class of the resulting model are still specified in a declarative manner. Declarative programming has many advantages. It renders concise problem specifications and it helps the user to avoid many of the errors that are common in imperative programming. There are, however, tasks which are cumbersome to specify declaratively while their corresponding imperative style solutions are much simpler. Therefore, a system allowing the combination of modules using imperative style programming with declarative attribute grammar modules would be desirable.

Our first paper uses a tool that supports textual modularization of RAGs and it is demonstrated how this facilitates and supports reusing and extending specifications. The second paper stresses the advantage of modularization from an aspect-oriented perspective and also the combination of modules written in imperative as well as declarative styles. The techniques have been implemented in a practical tool for RAGs, JastAdd.

Extensions of RAGs As has been mentioned earlier, RAGs facilitate tasks within traditional compiler-related applications. One example is name and type analysis of languages with complex scope rules. RAGs have also proven useful in a number of applications outside the traditional application areas of attribute grammars. An interesting question is then to what extent further extensions enhances the expressiveness of attribute grammars and facilitates tasks for the grammar author.

The second paper focuses on conceptual extensions such as the object-oriented view of attribute grammars and the combination of imperative and declarative programming code. A practical tool, JastAdd, that incorporates the extensions has been developed and tested.

In the third paper we deal with a formal extension: circular reference attributed grammars (CRAGs) and show how this widens the application area and makes specifications easier to write for many problems. The JastAdd tool has been enhanced to deal with circular dependencies and thereby we have been able to test the extended formalism in practice.

The fourth paper introduces further extensions: collection attributes and the combined formalism, circular collection attributes. Collection attributes are especially useful to specify solutions to “whole program problems”, i.e., problems where information from, potentially, the whole program must be combined. Solutions for these problems can also be specified using the classical AG formalism or RAGs, but using collection attributes yields much more concise specifications. For non-circular collection attributes our pro-

posed evaluation techniques also yield substantially faster evaluation than solutions based on ordinary attributes.

Evaluator implementation In Section 2 we mentioned that some of the evaluation techniques for traditional attribute grammars can easily be adapted to handle also circularities while others can be used for the RAG formalism. The question is then what evaluation technique is suitable for combined formalisms and how fast it is for practical applications.

In paper 2, we describe the basic demand-driven algorithm used for RAGs and show how it can be implemented in a very simple way in Java. In paper 3 this basic algorithm is extended to deal with the circular dependencies that may occur in CRAGs. A large part of paper 4 is devoted to different evaluation techniques for collection attributes, noncircular as well as circular.

The rest of this section briefly introduces the papers included in the thesis.

3.1 Paper 1: Program Visualization using Reference Attributed Grammars

The traditional application area for attribute grammars is related to compiler construction. One of our objectives has been to explore new application areas. This paper describes how RAGs can be used to integrate program visualization in language-based environments and how it can be specified and generated from grammars. It is shown how a general solution for a simple grammar can be reused in grammars for other specific languages.

As our experimental platform, we used an interactive language development tool APPLAB [5] that has been developed earlier at our department. The tool supports interactive development of application-specific languages. It is based on structure-oriented editing and makes use of RAGs. The user can organize the RAG specifications in several modules, thus separating different grammar aspects.

It is described how a reusable visualization specification can be obtained by using the modularization concept. Ideally, the visualization can be organized in three parts (each of which might be separated into modules). One part, the visualization front-end, captures the essence of the visualization by introducing node classes matching the main concepts of the visualization. The second part, the visualization back-end, then ties the first part to a certain visualization tool by specifying attributes to generate the representation required by the tool. The third part ties the front-end to a specific language. This part, the visualization glue module, can make use of static-semantic modules for the language at hand. The paper exemplifies the technique by using a state transition language as a running example. If, for example, a new state transition language is to be visualized, only the glue module has to be rewritten. The visualization front-end can be reused for all state transition languages and all visualization tools. Likewise, if a new visualization tool is to be used only the visualization back-end part needs to be rewritten.

The essence of the visualization specification is facilitated by reference attributes. For state transition languages, for example, the visualization is based on a state-transition graph which can be directly modelled as a RAG by tying nodes of the AST representing states to each other by reference attributes according to the transitions declared in the program at hand.

The work on which this paper is based was done before the implementation of the tool JastAdd, described in the second paper. Some of the specifications for the visualization were difficult to express in the declarative paradigm while their corresponding imperative implementations would have been simpler. Specifying program visualization would thus have been facilitated if combining the two paradigms had been possible. This observation supports one of the conclusions of the second paper, namely the advantage of combining imperative and declarative code.

The front-end of the visualization builds the graph on which the visualization is based. This is in essence realized by introducing two set-valued attributes in the node class corresponding to a vertex of the graph. For a particular vertex, one of them (`outgoingTrs`) models the set of target vertices for its outgoing edges and the other (`incomingTrs`) the set of source vertices for its incoming edges. The attributes are specified as traversals of the AST, assembling the proper nodes for each set. As will be described in the fourth paper, JastAdd has later been extended to support the collection attribute mechanism. The `outgoingTrs` and `incomingTrs` attributes are in fact excellent examples of when to use of collection attributes in order to yield simpler specifications. Both are examples of "whole program problems" since they combine information from, potentially, the whole program. More precisely, every node instance in the AST of the type corresponding to an edge of the visualization graph is a potential contributor to the information assembled in `outgoingTrs` and `incomingTrs`. By declaring them as collection attributes the explicit tree traversal is no longer needed. Instead, contributions are concisely specified in the AST class corresponding to edges as shown in Fig. 2.

3.2 Paper 2: JastAdd - an aspect-oriented compiler construction system

The paper describes JastAdd, a Java-based system for compiler construction built on top of the JavaCC parser generator [2]. The tool is centered around the object-oriented representation of the AST and supports modularized compiler implementation.

Usually, an abstract grammar is only a simplification of the corresponding parsing grammar leaving out tokens that do not carry semantic values and extra nonterminals introduced to resolve parsing ambiguities. In many cases, however, it is useful to impose different structures in the abstract and parsing grammars for some constructs. In JastAdd, the user specifies the abstract grammar independently

```
class StateDecl{
  coll HashSet incomingTrs() [new HashSet()] with add;
  coll HashSet outgoingTrs() [new HashSet()] with add;
}

TransitionDecl contributes
destState()
to StateDecl.outgoingTrs()
for sourceState();

TransitionDecl contributes
sourceState()
to StateDecl.incomingTrs()
for destState();
```

Figure 2: Yielding a more concise front-end by using collection attributes

of the underlying parsing grammar. JastAdd uses JavaCC and its underlying tree-building system JJTree for parser construction but its design is not tied to JavaCC. The abstract grammar in JastAdd is object-oriented (see Section 1.5) and defines a class hierarchy augmented with subcomponent information corresponding to production right-hand sides.

Different aspects of a compiler can in JastAdd be specified in separate modules. In imperative style modules (jadd-modules) fields and methods can be added to different node classes introduced by the abstract grammar. These modules use ordinary Java syntax. In declarative modules (jrag-modules), attributes and their equations can be added to the node classes. These modules use a somewhat extended Java syntax. The jrag-modules are translated into a jadd-module by the tool. The translated module implicitly defines a demand-driven evaluator for the attribute grammar of the jrag modules implemented as fields and methods. JastAdd generates node classes according to the abstract grammar and weaves into each node class the additions made in all the different jadd-modules (one of which might be a translation of jrag-modules).

We have quite substantial experience of using JastAdd both in education and in research. The combination of object-oriented ASTs, aspect modularization and the capability of combining imperative and declarative code has proven very useful. Other systems for RAGs and similar formalisms (for example APPLAB used in the previous paper) often have their own formal languages for specification. JastAdd, in contrast, is based on Java which makes the system easily accessible for many users.

The paper presents the original version of JastAdd, supporting RAGs. Its implementation uses JavaCC visitors. It has later been re-implemented by Torbjörn Ekman. The new implementation is bootstrapped using RAGs and has AspectJ-like notation for intertype declarations. Many features and extensions were added. Extensions include AST rewrites [11] and higher order attributes [9]. The new

JastAdd version also supports various shorthands. For example, imperative style code and code using the extended formalism for attribute grammars can now be written in the same module. The newer JastAdd is also independent of JavaCC and can be used together with other Java-based parser generators.

Further extensions have later been added: Circular reference attributed grammars as described in the third paper of this thesis and the collection attribute mechanism as described in the fourth paper.

The current version of JastAdd [1] has been used for implementing a complete compiler for Java [10]. This extensible Java compiler is used in the fourth paper for experimenting with collection attributes for large Java programs.

3.3 Paper 3: Circular Reference Attributed Grammars – their Evaluation and Applications

In traditional attribute grammars, all direct dependencies between attributes must be local involving only attribute instances of AST nodes of one production. As has been mentioned before, reference attributed grammars, RAGs, lift this restriction. RAGs therefore facilitates, e.g., the task of specifying name and type analysis for languages with complex scope rules. Many problems include name analysis as a subproblem on which further analyses can be built. Their specifications are, as a consequence, also facilitated by RAGs.

Some researchers have pointed out that allowing circular dependencies between attributes (under proper constraints to guarantee that the grammar is well defined) makes it easy for the AG author to specify problems that are naturally solved using mathematical recursion. In many cases the recursive solutions can be directly translated into a circular attribute grammar.

In the third paper we propose the combined formalism circular reference attributed grammars, CRAGs. We show how an evaluator for CRAGs can be automatically generated. We also explore the expressiveness of CRAGs by application examples. They include classical examples for CAGs as well as problems from new areas.

Our compiler construction tool JastAdd, described in the previous paper, is used as the experimental platform. Its evaluator generator capability has been generalized to include the necessary iterative process for circularly defined attributes. For performance reasons, as well as for robustness, the evaluator code for non-circular attributes has also been modified.

Two classical examples for circular attribute grammars are revisited: live analysis in optimizing compilers and the analysis of languages where constants can be used before declaration. In the first case we show that a larger class of languages can be handled by CRAGs given the possibility to use reference attributes. In the second case we show that reference attributes make it possible to specify the solution in a straight-forward way without introducing any circular dependencies.

We also exemplify the applicability of CRAGs by a highly recursive problem: the computation of *nullable*, *first* and *follow* used in parser construction. This is a problem that to our knowledge has not been solved using attribute grammars before and is typical for a large class of problems dealing with properties of grammars, so called grammar flow analysis [19, 25]. The specification of these computations clearly shows how the mathematical definitions of these concepts can be almost directly formulated as an attribute grammar. The task of computing the fixed points is the responsibility of the evaluator and is completely hidden from the AG author.

Many problems include name analysis of some kind as a subproblem and many analysis problems are inherently circular and need to be computed by iterating to a fixed point. We therefore expect CRAGs to be useful for a number of practical applications. We have also compared our demand-driven evaluator with hand-written imperative code implementing fixed-point iterations (in JastAdd). These experiments indicate that, for larger applications, solutions based on CRAGs are somewhat faster.

3.4 Paper 4: Collection Attribute Algorithms

In name analysis we introduce reference attributes in nodes corresponding to use sites in the AST. These attributes are specified to reference the corresponding declaration sites and are used, for example in type analysis, to propagate information from the referenced (declaration) sites to referencing (use) sites. For a certain metrics problem it might be interesting to find all use sites of a particular declaration. I.e., there is a need for information to flow from referencing (use) sites to referenced (declaration) sites.

Similar cross-referencing problems include finding all calls of a method, all subclasses of a class, and all overriders of a method. These problems are whole program properties in the sense that the cross references might be located in practically any part of the program.

Combined properties, like those mentioned above, can be modelled by ordinary synthesized and inherited attributes. However, the use of *collection attributes* as defined by Boyland [6], makes their specification much easier and more concise.

In the paper we describe how the collection attribute mechanism has been introduced in JastAdd and discuss a variety of possible evaluation techniques. The techniques are compared with respect to applicability and performance. The pure demand-driven techniques are shown to be outperformed by other alternatives which are not purely demand driven.

We also compare solutions based on collection attributes to those using ordinary attributes with respect both to simplicity and performance. We demonstrate how collection attributes raise the abstraction level of RAGs and yield much more concise specifications. They are also shown to open up for more efficient implementations.

A number of applications are presented as are experimental results for using collection attributes for their solutions. Examples include devirtualization and metrics problems for Java programs with 100k lines of code.

The paper also presents the combined formalism *circular collection attributes* and a number of algorithms for their evaluation. One of the examples from paper 3, dealing with computation of *nullable*, *first*, and *follow* is revisited to demonstrate how circular collection attributes facilitate specifications. More precisely, it is shown how the specification of *follow* becomes much shorter and simpler when circular collection attributes are available.

4 Contributions and Future Work

The main contributions of this thesis are connected to the experience of applying combined extended AG formalisms to different application areas. This work has included the development of a tool, JastAdd, which incorporates formal as well as conceptual extensions of AGs: reference attributes, circular dependencies, collection attributes, an object-oriented view of the grammar, and modularization concepts. It also allows the user to separate the abstract grammar from the parsing grammar and to combine imperative implementation code with attribute grammars. As a part of the JastAdd implementation, an evaluator generation technique capable of handling the combined formalisms was developed.

In the following subsections we summarize our contributions and discuss some possible directions for future work.

4.1 Contributions

In this subsection the contributions of the research presented in the papers are summarized and related to the list of objectives given in Section 3. The contributions of the author of this thesis is summarized in the last part of this subsection.

Paper 1 We, as well as other researchers [8, 26], have exemplified new areas, outside the traditional compiler-related ones, where RAGs can be applied. The program visualization application described in paper 1 is one example of how RAGs can be used outside the traditional compiler-related area for AGs.

The paper also demonstrates the advantages of modularization. The object-oriented view of the grammar is combined with the possibility to separate specifications in modules according to different aspects of the problem.

Based on our example applications, we conclude that the combination of RAGs with a modularization concept supports separation of concerns and makes it easy to understand and also to reuse and extend the specifications.

Paper 2 The advantages of modularization techniques are again stressed in this paper. The concept of modularization is generalized to also include the possibility of combining declarative attribute grammar modules with imperative style modules using ordinary Java syntax. The most appropriate technique for each subproblem can thus be used.

Even in extended formalisms, there are some computations that do not lend themselves easily to declarative specification, while they are trivial to express using imperative style programming. Our conclusion is therefore that the possibility to combine imperative and declarative aspects is very useful. It is shown how the generalized modularization technique can be implemented in a Java-based system. This includes the implementation of a demand-driven evaluator for the attribute grammar modules. The construction of the evaluator is straight-forward, given the object-oriented view of the grammar. Furthermore, the demand-driven technique facilitates the integration of attribute grammar modules and imperative modules. Attributes defined in declarative aspects can be accessed by imperative aspects. The access of an attribute causes the demand for its evaluation.

In our experience, the combination of declarative and imperative modules is very useful. JastAdd has been used quite extensively in research projects at our department and also in education.

Paper 3 Circular Reference Attributed Grammars (CRAGs) are introduced in this paper. An important contribution of the paper is the development of algorithms for evaluation of CRAGs. A CRAG is a combination of two extensions of AGs; allowing attributes to be references to nodes in the AST and allowing circular dependencies between attribute instances under proper constraints. The results described indicate that CRAGs have a number of practical application areas and significantly widens the application area of AGs. It is exemplified by the computation of *nullable*, *first* and *follow* introduced in the context of parser construction. This problem is representative for a large class of so called grammar flow problems, which to our knowledge has not been specified using the attribute grammar formalism before.

Many problems have solutions that can be expressed using mathematical recursion including circular dependencies. We demonstrate how these solutions can be almost directly formulated as CRAG specifications.

Furthermore, CRAGs widen the scope of specifications for some classical problems where CAGs have been used before. Live analysis in optimizing compilers is one example. This is a classical example used to demonstrate the usefulness of CAGs. In a CRAG, where attributes are allowed to reference remote nodes in the AST, the specification can be generalized to handle a larger class of languages. An evaluator for CRAGs has been implemented by generalizing the evaluator construction mechanism in our tool JastAdd.

We have found that the demand-driven technique can be adapted to handle CRAGs. We have compared our demand-driven evaluation algorithm with handwritten imperative code implementing fixed-point iterations. The results indicate that there is little difference in performance. Solutions based on CRAGs even seem to be somewhat faster for large applications.

Circular attributes are used in the extensible Java compiler implemented in JastAdd [10], for example, for checking that inheritance hierarchies are acyclic.

Paper 4 Collection attributes, non-circular as well as circular, are introduced. We demonstrate through application examples how these extensions further raise the abstraction level of AGs. Important contributions of this paper are the development of a number of evaluation algorithms for the extended formalisms and a comparison of the algorithms with respect to performance.

It is shown how the extended formalism can be used to specify concise solutions to problems in different application areas.

It is also shown how the AG evaluator can be enhanced to handle the extended formalisms in different ways. In particular it is shown how small deviations from purely demand-driven techniques improves efficiency substantially.

Contributions by the author

The author of this thesis is the main author of papers one, three and four. For all papers included in the thesis, the work on presentation has been shared among the authors.

Concerning the work described in the papers, the author of this thesis contributed as described below:

Paper1 Design, specification and implementations of the solutions described.

Paper 2 Implementation of the attribute grammar part of JastAdd.

Paper 3 Construction of all evaluation algorithms for circular attributes and their implementation in JastAdd. Implementations of all examples in the paper. Analysis of performance through experiments.

Paper 4 Construction of all evaluation algorithms for collection attributes and their implementation in JastAdd. Analysis of applicability and performance for these algorithms. Implementation of all examples except the metrics example, which was implemented by Torbjörn Ekman.

4.2 Future work

There are many interesting ways in which this research may be continued:

Modularization An interesting field for future research is a generalization of the modularization concept to further support reusability. For example, in the first paper of this thesis, the visualization modules are not completely reusable. They specify a graph-based visualization by adding attributes and equations to node classes corresponding to vertices and edges. In the solution presented in the paper, they rely on the underlying grammar to use the same name for these node classes. In an extended experimental version of JastAdd, we are currently investigating how to overcome this problem. One idea is to allow aspects to introduce interfaces containing declarations of AG attributes and also to specify additions to such interfaces. An AST class can declare that it implements such an interface. The tool then checks that it provides specifications of the interface attributes. Also, all additions to the interface specified in aspects will be added to all classes implementing the interface. Using these features for the visualization example, the glue module can be replaced by interfaces. One interface represents vertices of the graph and another interface represents edges. These interfaces are declared to contain the same attributes as the glue module of the present solution. These attributes are used in the visualization modules, where specifications now can be expressed as additions to the interfaces instead of additions to certain AST node classes.

Evaluation of RAGs The evaluation technique we use for RAGs can probably be improved in several ways. One possibility would be to support automatic caching of attributes. The tools used in our work, APPLAB and JastAdd, both allow the user to declare which attributes to cache. Optimal evaluation is, in principle, achieved by caching all attributes. Avoiding caching of attributes that are accessed only once will, however, decrease memory usage, and improve performance in practice. It would be desirable to develop a technique for automatically deciding which attributes to cache for best performance and memory usage either based on static analysis of the grammar, or on profiling, or on a combination of these.

Interaction with rewrites There are issues concerning the interaction between the evaluation of circular attributes and tree rewrites [11] that need to be further investigated. For example, it is desirable to cache circular attributes since their evaluation is potentially expensive. Taking rewrites into consideration, it is not possible to cache an attribute instance if it is dependent on attributes instances in nodes that are not final, i.e., that can be subject to further replacements. There are several cases that need to be considered concerning, for example, whether the iterative process starts in rewrite mode or not, whether attributes of non-final nodes are needed during iterations or

not and so on. In some of these cases, circular attributes must not be cached. Instead, they must be set back to their bottom values. These cases can be detected by the JastAdd tool and there is already an implementation in place for some of them. Further interference scenarios must, however, be investigated. It is also important to find out to what extent these situations appear in practical applications.

Incremental evaluation Incremental evaluation techniques have been developed for classical AGs [28] and for CAGs [17]. Boyland [7] developed an incremental technique for a RAG-like formalism that has been tried out on small example grammars. It is a very interesting and challenging question for further research to explore the possibility to develop incremental evaluation techniques for CRAGs and for collection attributes. Incremental techniques for these combined formalisms, capable of handling large programs, would be of great practical interest. Examples include Java compilers used in integrated development environments.

Continued work on CRAGs Future work also includes improving the CRAG evaluator. As is pointed out in paper 3 the evaluator does not always detect that circularly dependent attribute instances belong to different strongly connected components of the dependency graph. As a result, iterations are sometimes performed over more than one component at a time, resulting in suboptimal performance and requiring monotonicity over the combined components. It might be possible to improve component detection by using the underlying modularity of the specifications.

Another issue concerns detecting circularities. In the present version of JastAdd the user must declare which attributes are circular. The tool will detect undeclared circularities during evaluation and treat these as exceptions which cause termination of the evaluation process. It would be desirable to have a tool that detects all possible circularities before evaluation and then generates an evaluator that performs the necessary iterations for the detected possible cycles. It is, however, an open question to what extent it is possible to detect possible circular structures statically for RAGs. Such analysis would need to be conservative and it would be interesting to look into possible approaches and their applicability.

Application areas We also plan to explore new application areas for CRAGs and for collection attributes. For CRAGs, looking into grammar flow analysis work [19,25] and different types of data flow analysis are of special interest. For the combined formalism with collection attributes we have only begun to explore the possibilities, and we believe that there are many more interesting applications areas than those presented in the fourth paper.

BIBLIOGRAPHY

[1] JastAdd, 2007. Available at: http://www.jastadd.org.

[2] The JavaCC Project, 2007. Available at: http://javacc.dev.java.net.

[3] B.Courcelle and P. Franchi-Zannettacci. Attribute grammars and recursive program schemes. *Theoretical Computer Science*, 17:163–191, 1982.

[4] G. M. Beshers and R. H. Campbell. Maintained and constructor attributes. In *Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments*, pages 34–42. ACM Press, 1985.

[5] E. Bjarnason. Interactive Tool Support for Domain-Specific Languages, December 1997. Licentiate Thesis. Lund University, Sweden.

[6] J. T. Boyland. *Descriptional Composition of Compiler Components*. PhD thesis, University of California, Berkeley, 1996.

[7] J. T. Boyland. Incremental evaluators for remote attribute grammars. *Electronic Notes in Theoretical Computer Science*, 63(3), 2002.

[8] A. Cornils and G. Hedin. Tool support for design patterns based on reference attributed grammars. In *Proceedings of WAGA'00, Workshop on Attribute Grammars and Applications*, 2000.

[9] T. Ekman. *Extensible Compiler Construction*. PhD thesis, Lund University, Sweden, 2006.

[10] T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. Accepted for publication at OOPSLA'07.

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

[12] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of CC'86*, pages 85–98. ACM Press, 1986.

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

[14] G. Hedin. An Overview of Door Attribute Grammars. In *Proceedings of CC'94*, volume 786 of *LNCS*, pages 31–51, Edinburgh, 1994.

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

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

[17] L. G. Jones. Efficient evaluation of circular attribute grammars. *ACM Transactions on Programming Languages and Systems*, 12(3):429–462, 1990.

[18] M. Jourdan. An optimal-time recursive evaluator for attribute grammars. In *International Symposium on Programming*, volume 167 of *LNCS*, pages 167–178. Springer, 1984.

[19] M. Jourdan and D. Parigot. Techniques for improving grammar flow analysis. In *European Symposium on Programming*, volume 432 of *LNCS*, pages 240–255, 1990.

[20] G. E. Kaiser. *Semantics for structure editing environments*. PhD thesis, Carnegie Mellon University, 1985.

[21] U. Kastens. Ordered Attribute Grammars. *Acta Informatica*, 13(3):229–256, 1980.

[22] T. Katayama. Translations of attribute grammars into procedures. *ACM Transactions on Programming Languages and Systems*, 6(3):345–369, 1984.

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

[24] O. L. Madsen. On defining semantics by means of extended attribute grammars. In *Semantics-Directed Compiler Generation, Proceedings of a Workshop*, volume 94 of *LNCS*, pages 259–299. Springer, 1980.

[25] U. Möncke and R. Wilhelm. Grammar flow analysis. In *Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 151–186, 1991.

[26] P. Persson and G. Hedin. Interactive Execution Time Predictions Using Reference Attributed Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 173–184, Amsterdam, 1999.

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

[28] T. Reps and T. Teitelbaum. The synthesizer generator. In *ACM SIGSOFT-/SIGPLAN Symp. on Practical Software Development Environments*, pages 42–48. ACM Press, 1984.

INCLUDED PAPERS

PAPER I

PROGRAM VISUALIZATION USING REFERENCE ATTRIBUTED GRAMMARS

Abstract

This paper describes how attribute grammars can be used to integrate program visualization in language-based environments and how program visualizations can be specified and generated from grammars. It is discussed how a general solution for a simple grammar can be reused in grammars for other specific languages. As an example we show how diagram generation for a very simple state transition language can be integrated in a more complex specific state transition language. We use an extended form of attribute grammars, RAGs, which permits attributes to be references to nodes in the syntax tree. An external graph drawing tool is used to visualize the diagrams. The solution is modularized to support reuse for different languages and exchange of the external drawing tool for different types of visualization.

Published as: E. Magnusson and G. Hedin. “Program Visualization using Reference Attributed Grammars". Nordic Journal of Computing 7 (2000) 67-86.

1 Introduction

Program visualization is an important technique useful to gain understanding of the structure of a program. Meaningful visualizations can be built from several different types of elements (words, images, ..) but graph drawing is the most popular way to present structural relationships. One example is call-graphs where functions are presented as nodes in a directed graph and possible calls between functions as edges. Other examples are UML class diagrams, where design is expressed through graphical notations, and state transition diagrams used to visualize finite state machines.

In this paper we discuss integrating program visualization in language-based environments and how such program visualizations can be specified and generated from grammars. We deal with static code visualizations, i.e., visualizations of the program code, in contrast to, e.g., dynamic code visualization (visualizations of an executing program) and algorithm visualization. We have an interactive environment, APPLAB (APPlication language LABoratory) supporting language-based editing of the grammars of a language as well as language-based editing of programs in the language [5–7]. APPLAB is based on structure-oriented editing and reference attributed grammars [14] (an object-oriented extended form of attribute grammars).

The main representation of the program is an abstract syntax tree (AST), but a program visualization is often based on some kind of graph. We use reference attributed grammars to describe how these graphs can be generated from the syntax tree. External visualization tools can then be integrated provided that they have an import mechanism for graphs from text files in some documented format. The generation of the text files on the format required by the tool is also specified using reference attributed grammars.

We show how it is possible to modularize the solution for a particular kind of visualization so that both the underlying programming language and the external visualization tool can easily be exchanged. In this article, we use state-transition visualizations as a running example and show how the solution can be reused for different state-based languages. A similar technique could be used for other visualizations, e.g. to obtain UML class diagrams for different object-oriented programming languages.

The rest of this article is organized as follows. Section 2 presents the environment architecture. Section 3 describes a general solution for a simple state transition language and section 4 gives an overview of how the representation for an external tool can be generated. In section 5 we show how the solution can be integrated in a more complex specific state transition language. Comparison of our approach to some related work is done in section 6. Section 7, finally, gives a concluding discussion of our technique and how it can be developed further.

2 Environment architecture

The environment architecture consists of two parts; a language environment supporting language specification and language-based editing of programs in the specified languages, and a visualization tool, i.e., an external graph drawing tool used for visualizing programs. In our experimental platform we use our interactive language development tool APPLAB as the language environment, and the tool daVinci [1] from University of Bremen as the visualization tool. See Fig. 1.

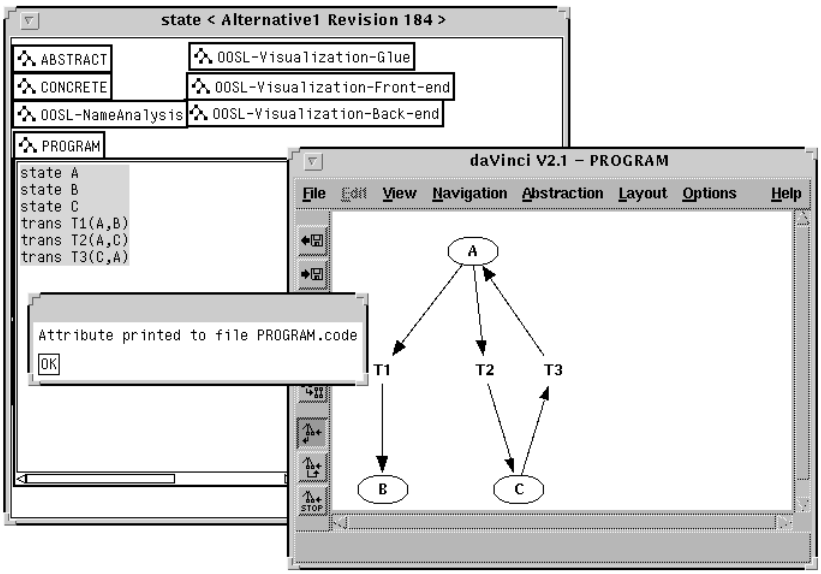

Figure 1: Overall architecture. To the left, the language environment APPLAB with the language specification in several aspects (icons ABSTRACT, CONCRETE, OOSL-Name-Analysis, ...) and an example program in the specified language (window PROGRAM). To the right, the daVinci tool is used to visualize the program as a state-transition diagram.

2.1 The interactive language development tool APPLAB

The language environment is implemented using our interactive tool APPLAB (APPlication language LABoratory) [5, 6]. The main goal of APPLAB is to support interactive development of application-specific languages, allowing the user to simultaneously work on the language definitions and experiment with the result-

ing language. Changes in the language are immediately reflected in the program editor. The system is based on structure-oriented editing and an object-oriented extension to attribute grammars.

An attribute grammar [15] is an extension of a context-free grammar where a set of attributes $A(X)$ are associated with each non-terminal symbol X. A set of equations $E(p)$ are associated with every production p. There are two kinds of attributes, synthesized and inherited. Synthesized attributes are used to propagate information upwards in the syntax tree and inherited attributes to propagate information downwards. For each production $p : X_0 ::= X_1 \ldots X_n$, $E(p)$ should have equations defining all the synthesized attributes of X_0 and all the inherited attributes of $X_i, i = 1, 2, \ldots n$.

In the extended attribute grammars used in APPLAB, the context-free syntax is modeled as an object-oriented inheritance hierarchy of node classes, where the leaf classes correspond to productions and their superclasses to nonterminals. The class hierarchy allows attribute grammars to be written in a compact way by using the inheritance hierarchy to avoid much of the repetition of attributes and equations that is otherwise common in classical attribute grammars [12].

The root of the class hierarchy is the node class `ANYNODE` which can be used to model behavior common to all nodes in the AST. Every node in the syntax tree is an instance of a subclass of `ANYNODE`. Equations defining attributes can be viewed as parameterless virtual functions. It is therefore possible to make a default definition of an attribute in a superclass and then override it in a subclass. APPLAB also supports the definition of virtual functions with parameters.

APPLAB makes use of object-oriented concepts to organize the specification, but in contrast to object-oriented programming languages the specification contains only declarative constructs (no assignments or other imperative constructs). The attribute evaluation method used is demand evaluation, a simple but general evaluation method based on recursion [17]: When an attribute value is demanded, the right-hand side of its defining equation is evaluated (similar to a function call) and this will in turn lead to the demand evaluation of the attributes used in that equation. Optimal evaluation is achieved by caching evaluated attributes in the AST and cyclic definitions of attributes can be detected at evaluation time by setting a flag for each cached attribute. In APPLAB, the user can demand an attribute value to be displayed or written to a file.

In addition to the object-oriented style of specifying the attribute grammar, APPLAB supports reference attributed grammars, i.e., the ability to let an attribute be a reference to an arbitrary node in the syntax tree [14]. Reference attributes are similar to ordinary reference variables in object-oriented programming languages in that they can refer to other objects and be used to obtain arbitrary linked data structures (including cyclic structures). However, reference attributes differ from reference variables by being defined declaratively by equations. This is in contrast to the usual programming language approach of writing an imperative mutating computation to obtain the linked structure.

Reference attributes are useful for describing arbitrary relations between nodes in a syntax tree, in addition to the syntactic (tree-structured) relations that ordinary attribute grammars support. For example, call graph relations, inheritance relations, and state-transition relations are easily described using reference attributes. A few built-in structured data types like dictionaries mapping strings to node references (NodeDictionary) and sets/bags of node references (NodeBag) have been added to APPLAB in order to allow such relations to be described effectively.

A language is specified in APPLAB in a document containing several grammar aspects, see also Fig. 1. The ABSTRACT aspect defines the abstract context-free syntax of the language and the CONCRETE aspect defines the concrete syntax (how to unparse an AST as text in a window). The OOSL (Object-Oriented Specification Language [5, 6, 13]) aspect defines an attribute grammar. An OOSL specification can be split in a number of modules thus textually separating attributes and equations for different purposes, e.g., static-semantic checking and code generation. Similar possibilities for modularizing the attribution specification is available also in non-object-oriented attribute grammar systems such as the Synthesizer Generator [20]. From an object-oriented viewpoint, the OOSL modules are orthogonal to the class hierarchy. Similar modularization techniques are available also for some object-oriented programming languages, in particular the fragment system for the BETA language [16] and in subject-oriented programming [11].

Fig. 2 shows an OOSL example of how to add an inherited attribute `root` referencing the root node in the AST to every node. The equation is an example of a so called collective equation which defines the value of an inherited attribute of all sons of a given type, in this case of any type [12]. The example also shows an example of an overriding equation: the equation in `Program` overrides the default definition in `ANYNODE` since `Program` is a subclass of `ANYNODE`.

Non-terminal	*Attributes*	*Equations and functions*
ANYNODE	inh root: ref Program	eq son ANYNODE.root := root
Program		eq son ANYNODE.root := this Program

Figure 2: Adding an inherited reference attribute `root` to all nodes. The default definition in `ANYNODE` defines the `root` attribute of all son nodes to be the same as for the current node. In `Program` (the root production), the definition is overridden, defining the `root` attribute of all sons to the `Program` node to be a reference to the `Program` node.

2.2 The graph drawing tool daVinci

The external tool used for the graph visualization is daVinci [1], an interactive visualization system for drawings of directed graphs, developed at the Computer Science Department at the University of Bremen. An application program can access the operations of daVinci by using its API. The communication between daVinci and the application is realized with UNIX pipes. There is also a Graph Editor Application which is an interactive tool to create and modify graphs. The editor in this case acts as an application program which communicates with the daVinci API. Currently, we use the daVinci editor only to display graphs, not to edit them. Graphs can be loaded in the editor from text files with a special format, the term representation, described in more detail in Section 4.1.

After loading a graph into daVinci it can be processed in different ways. For example edge crossing minimization and edge bending minimization can be performed. It is also possible to create a survey view of the graph and zoom into different part of it and to change the orientation of the graph.

In our experimental platform, an attribute grammar is specified in APPLAB which defines the representation required by daVinci as a string attribute. Thus, to visualize a program, this attribute is evaluated and saved on a text file which is then loaded into daVinci, and displayed on the program window of the daVinci editor. See Fig. 1. Our ambition is to improve the integration of the language environment and the visualization tool. Preferably it should be possible to connect to the external graph drawing tool directly from APPLAB.

2.3 Obtaining a reusable visualizatation specification

In order to obtain a general reusable solution, it is useful to organize the specification of a visualization according to the following different aspects: First, the essence of the visualization can be specified by introducing node classes that match the main concepts in the visualization. For example, for a state diagram, the node classes could be `State` and `Transition`. Attributes of these node classes are introduced for modelling the essential properties of the visualization. We call this part of the specification the *visualization front-end*. Second, to tie this specification to a certain visualization tool, a *visualization back-end* module is introduced which specifies the computations needed to generate the representation required by the external visualization tool. If the visualization tool is exchanged, another back-end is written for that tool. Third, a *visualization glue* module is written which ties the front-end to the specific language to be visualized. Typically, the glue module can make use of a static-semantics module which defines the name analysis (identifier declaration/use sites) for the language at hand. To visualize another language with the same kind of diagrams, a new glue module is written. The front-end module can be reused for all languages and visualization tools. This module organization is shown in Fig. 3.

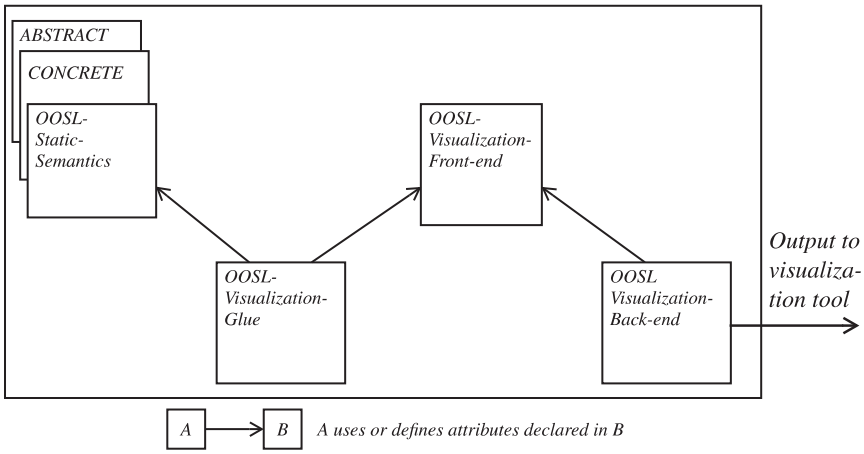

Figure 3: Internal architecture in APPLAB showing module dependencies for a general reusable specification of the visualization.

APPLAB currently supports this module organization, with the restriction that the front-end must use the node classes that correspond to states and transitions in the ABSTRACT syntax. As future work, we plan to generalize the module system of APPLAB in order to allow the front-end to introduce its own node classes, and let the glue module tie these node classes to the corresponding ones appearing in the ABSTRACT syntax.

3 Visualization for a simple state transition language

We will use state-transition diagrams as our running example. In this section, we introduce a very simple state transition language, TinyState, and the front-end and glue of our solution, i.e., how to specify the essence of the state-transition graph on which visualizations of programs written in TinyState are based, and how to tie this to the syntax of TinyState.

3.1 The language TinyState

The abstract grammar for a very simple state transition language, TinyState, is given in Fig. 4. Production (2) is a list production stating that a `StateDecls` consists of a number of `StateDecl` nodes. Production (5) is a construction pro-

duction stating that `TransitionDecl` is a construction of three IDs (the name of the transition, the name of the source state and the name of the target state).

```
Program ::= StateDecls TransitionDecls   (1)
StateDecls ::=StateDecl*                 (2)
StateDecl ::= ID                         (3)
TransitionDecls ::= TransitionDecl*      (4)
TransitionDecl ::= ID ID ID              (5)
```

Figure 4: The ABSTRACT grammar of TinyState, a simple state transition language.

The concrete syntax of TinyState becomes evident from the example program of Fig. 5. The program can be visualized as a directed graph, where vertices correspond to states and edges to transitions.

3.2 Visualization front-end

The visualization front-end defines a representation of the state-transition graph that is independent of the programming language syntax and which is easy to traverse for the back-end. The representation is realized using reference attributes that link together declarations of transitions and declaration of their source and target states.

In every `TransitionDecl` node we add two attributes `sourceState` and `targetState` referencing the `StateDecl` nodes in the tree corresponding to the source and target states. Every `StateDecl` node has an attribute `outgoingTrs` defined to be a set of references to the transitions having the actual state as its source. There is also a corresponding attribute `incomingTrs`

```
state a
state b
state c
trans t1(a,b)
trans t2(a,a)
trans t3(b,c)
trans t4(c,a)
```

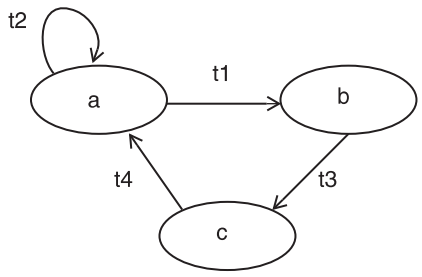

Figure 5: A simple program and a possible visualization.

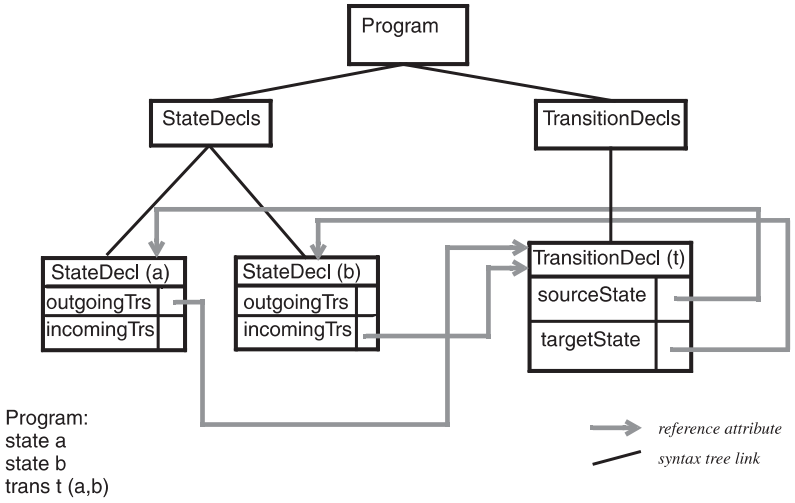

Figure 6: Connections between states and transitions for a small program.

for transitions having the state as their target. In this way we get a description of the graph resembling an ordinary adjacency list representation which makes it easy to traverse. In Fig. 6 the connections between `StateDecl` nodes and `TransitionDecl` nodes in the AST for a small program are shown.

To define the attributes `sourceState` and `targetState`, detailed knowledge about the programming language syntax is needed, and the equations defining these attributes are therefore placed in the glue module. The attributes `outgoingTrs` and `incomingTrs` can then be computed in a syntax-independent way, using the values of `sourceState` and `targetState`. E.g., `outgoingTrs` can be defined by searching the AST for `TransitionDecl` nodes where `sourceState` references the actual state. A function `outTrs` is defined which recursively searches the tree for nodes matching this condition. To be able to start the search at the root of the AST, a reference attribute `ASTroot` is introduced which must be defined by the glue module. The attribute `incomingTrs` is defined in a similar way. Fig. 7 shows the front-end module.

The `outTrs` function in Fig. 7 makes use of the *foreach*-construct in OOSL. In this case it is used to iterate over all the sons of an arbitrary AST-node. The same construct can be used to iterate over the elements in a NodeBag. (Despite its imperative appearance, the foreach construct is a declarative language construct equivalent to a special case of a tail-recursive function.) The function `outTrs` may seem unnecessarily complicated for our simple language. Since we know that all `TransitionDecl` nodes in the AST are sons of the `TransitionDecls`

Attributes to be defined in the glue module		
Non-terminal	*Attributes*	*Equations and functions*
TransitionDecl	syn sourceState: ref StateDecl; syn targetState: ref StateDecl; syn transitionLabel: string;	
StateDecl	syn stateLabel: string;	
ANYNODE	syn ASTroot: ref ANYNODE;	

Additional attributes and definitions		
Non-terminal	*Attributes*	*Equations and functions*
ANYNODE		outTrs: func ref NodeBag (n: ref StateDecl) foreach $X: ANYNODE in this ANYNODE do $N := (init new NodeBag) inspect $Y := $X when TransitionDecl do if $Y.sourceState=n then $N.add($Y) else $N otherwise $N.union($Y.outTrs(n))
StateDecl	syn outgoingTrs: ref NodeBag; syn incomingTrs: ref NodeBag	eq outgoingTrs := ASTroot.outTrs(this StateDecl) eq incomingTrs := ASTroot.inTrs(this StateDecl)

Figure 7: Front-end module. The function `inTrs` used in the definition of `incomingTrs` is similar to `outTrs` and not shown.

node iterating over these would be sufficient. Implementing this function (and others) in a more general way means, however, that we are able to reuse them when adding visualization aspects to other languages with completely different syntax tree structures.

We also declare two string attributes `stateLabel` and `transitionLabel` in the front-end module. They denote the text to be attached to nodes and edges respectively in the visualization graph and are to be defined by the glue module.

3.3 Static-semantics of TinyState

It is often useful to base the glue module on the static-semantics module, because this module already contains the name analysis needed for the glue module. The

static-semantic analysis of TinyState is very simple. One of its goals is to check that the names of states used in transition declarations are declared in the program. For this purpose an aggregate attribute `stateDict` (a NodeDictionary) is defined as a mapping from state names to references to their respective declaration nodes. Checking that a transition declaration is correct means checking that the names of the source and target states can be retrieved from the dictionary. The dictionary is an attribute defined in the root node of the syntax tree. All other nodes of the AST are given access to the dictionary via an attribute `root` referencing the root node and defined as was shown in Fig. 2. The definition of `stateDict` is shown in the table of Fig. 8.

Non-terminal	*Attributes*	*Equations and functions*
Program	syn stateDict: ref NodeDictionary	eq stateDict := a_StateDecls.buildDict()
StateDecls		buildDict func ref NodeDictionary := foreach $X : StateDecl in this StateDecls do $D := (init new NodeDictionary) $D.add($X.stateName, $X)
StateDecl	syn stateName: string	eq stateName := a_ID.val

Figure 8: Part of the static-semantics of TinyState. A table, `stateDict`, mapping state names to state declarations is defined.

The function `buildDict()` in node class `StateDecls` uses the *foreach* construct in OOSL. In this case a table is built containing association pairs (key, element) for all sons (all of which are `StateDecl` nodes) where key is the name of the son and element is a reference to the son node. If the same key is added more than once to a NodeDictionary the previous association is overridden. The table `stateDict` can therefore also be used when checking that all state names in a program are unique. To each `StateDecl` node an equation can be added where a lookup for the state name is performed. The corresponding element should refer to the actual state. If not, a violation of the uniqueness requirement has been detected. The complete static-semantics is available in a separate report [19].

3.4 The glue module

Fig. 9 shows the glue module. It should implement some of the attributes declared in the front-end according to Fig. 7. The attributes `sourceState` and `targetState` can be defined simply by doing a lookup in the dictionary `state-`

`Dict` defined in the static-semantics module. The `ASTroot` attribute can be defined simply using the `root` attribute (as defined in Fig. 2). For `stateLabel` and `transitionLabel` there is little choice in TinyState but to define them as the names of the corresponding state and transition respectively.

Non-terminal	*Attributes*	*Equations and functions*
TransitionDecl		eq sourceState := root.stateDict.lookup(a_ID_2.val) eq targetState := root.stateDict.lookup(a_ID_3.val) eq transitionLabel := a_ID_1.val
StateDecl		eq stateLabel := a_ID.val
ANYNODE		eq ASTroot := root

Figure 9: The glue module. The indexing on the identifiers (a_ID_1, a_ID_2 etc.) refers to the different occurences of the ID nonterminal in the `TransitionDecl` production. See Fig. 4.

4 Visualization back-end for the state transition visualization

Once the graph has been described in the front-end, by linking together `StateDecl` and `TransitionDecl` AST nodes via reference attributes, the representation required by the external graph drawing tool can be specified independently from the actual underlying language. In this section we give an example of such a back-end, by showing how code is generated for output to the daVinci tool. The daVinci tool represents graphs as nodes and edges, and the goal of the back-end is thus to map the `StateDecl-TransitionDecl` graph of the program to the format for nodes and edges required by daVinci. This mapping is non-trivial because daVinci represents graphs as trees with special treatment of edges that cannot be mapped to a tree and special treatment of cyclic structures in the graph.

4.1 The daVinci term representation

When graphs are loaded in daVinci a special format called the *term representation* is used. The term representation is defined by a context-free grammar. A term is a structure of type `parent[child`$_1$`, child`$_2$`,...]`. Brackets are used around a list of elements of the same type. The scheme is applied recursively. The term representation is plain text, so it can be created manually using a text editor.

Typically, however, it is created automatically by some application program. In our case the input to daVinci is created by defining a string attribute of the program to be visualized.

Identifiers and *references* are used to identify daVinci nodes and edges. If a child node has more than one parent the subgraph of the child appears only once in the term representation (as a child of one of the parents). This subterm is given an identifier, the identifier of the child node. The other parents have only a reference to this identifier. For example the node C in the graph of Fig. 10 has two parents A and B. When generating the code the node A will be treated as parent of C and when visiting C on a traversal coming from A we will continue recursively to generate the code of the subgraph of C. Coming from B, however, we will stop the traversal at C and just return the code of a reference to the child C.

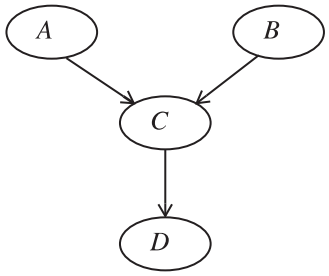

Figure 10: A node (C) with more than one parent.

One task of the back-end is to generate unique identifiers for all daVinci nodes and edges. This is done by adding an integer attribute `prefixNbr` to all nodes in the AST and defining this attribute as the number of the node when traversing the tree in prefix order starting with number 1 in the root node. The relationship between the number of a node and the number of its parent can be expressed as

```
prefixNbr:=parent.prefixNbr+1+nbrOfNodesInLeftSiblings
```

where the last term denotes the total number of nodes in the subtrees rooted in the left siblings of the actual node. This number can be computed using auxiliary functions implemented in `ANYNODE`, in a manner completely independent of the underlying programming language. Since daVinci requests its unique identifiers to be strings, another string attribute `nodeId` is defined in each node. Its value is simply the `prefixNbr` attribute translated into its corresponding string. See [19] for the specification of these computations.

The term representation uses attributes to specify the visualization of individual daVinci nodes and edges. We call these attributes *daVinci attributes* to distinguish them from attributes of an attribute grammar. All daVinci attributes have default values. The following example shows daVinci attributes for a graph node

which should be drawn as a box (a default shape value) with blue background and text "hello" written using the default font ("a" is the constructor for string pairs defining daVinci attributes in the term representation):

```
[a("OBJECT","hello"), a("COLOR","blue")]
```

All daVinci attributes that need to be defined have their corresponding attributes in our grammar. We have chosen to draw the nodes as ovals and edges as lines with an arrow pointing to the target state.

4.2 Code generation by graph traversal

The daVinci term representation (in the following called simply the code) can be generated by using information propagated along the reference attributes describing the graph, corresponding to depth-first graph traversal. There are however two problems which need to be dealt with.

The first problem concerns cycles in the graph. In an imperative language you usually perform depth-first traversal by adding an extra boolean attribute "visited", initially false and changed to true when visiting the node for the first time. In a declarative language it is not possible to change the value of an attribute once defined. The usual solution is to use sets to keep track of which nodes to visit next and which nodes to avoid to visit again. A simpler technique for the problem at hand is to avoid cycles by inverting edges and then draw them reinverted. Since all nodes in the AST are numbered (the `prefixNbr` attribute) we can introduce an order between states. We define a state `S1` to be declared before another state `S2` if `S1.prefixNbr < S2.prefixNbr`. An attribute `inverse` is added to `TransitionDecl` nodes. If the source of a transition is not declared before its target the value of `inverse` is true, otherwise false. In the code generation phase a transition where `inverse` is true will be treated as a transition from its target state to its source state. One of the daVinci attributes for edges specifies the way the edge should be drawn. If `inverse` is true then the corresponding daVinci attribute is defined to draw the edge inverted i.e. it appears with its original orientation in the visualization. Selfedges need special treatment.

When cycles are removed the code can be generated by appending the code representation of all `StateDecl` nodes with indegree 0. The code of a `StateDecl` is constructed by appending the code of all transitions in its `outgoingTrs` attribute where `inverse` is false and all transitions in its `incomingTrs` attribute where `inverse` is true. The code of a transition is in turn in principle the code of its target state. daVinci attributes are inserted in appropriate places as stated by the grammar of the term representation. Thus the generation of the code of a state corresponds to a depth-first traversal of the graph starting in the actual state.

The second problem originates from the requirement to describe a subgraph only once in the term representation as explained in section 4.1. For this purpose

we add an attribute `theTransition` to each `StateDecl` node defined to be a reference to one of the transitions having the actual state as its target. In the code generation we continue to recursively visit and generate the code of a target state node if the transition node being treated equals its `theTransition` attribute. Otherwise we just return the reference code (the unique identifier) of the target state node.

The complete back-end specification is available in [19].

5 Reusing the visualization specification for a more complex state transition language

The state-transition visualization specified by the front-end and back-end modules can be used for any state-transition language simply by exchanging the glue module. In another research project at our department, an application-specific language has been developed to support executable state-transition based specifications for devices communicating over short-distance radio. This project is done in cooperation with Ericsson Mobile Communications [10]. We have used this language, ExSpecState, as an example of how to integrate the diagram generation in a specific state transition language. In this section we will discuss the glue module for ExSpecState.

5.1 Differences between the languages

The ABSTRACT grammar of ExSpecSate contains approximately 90 productions. It comes equipped with OOSL grammar aspects for name and type analysis. As a state transition language it differs from TinyState in two ways. The states are hierarchical and the transitions have no names. An excerpt from the ABSTRACT grammar showing only the productions affecting state and transition declarations is given in Fig. 11. and part of a program using ExSpecState is shown in Fig. 12.

```
Root ::= Process*
Process ::= ID OptComment StateSpecification
StateSpecification ::= DeclList
DeclList ::= Decl*
Decl ::= StateDecl | TransitionDecl | VarDecl | ChannelDecl
StateDecl ::= ID OptFormalParamList OptCommentList OptStateSpecification
OptStateSpecification ::= NoStateSpecification | StateSpecification
TransitionDecl ::= EventDecl OptCommentList OptLocalDecls OptActionList Use
```

Figure 11: Some of the productions for the ExSpecState language.

```
process MP: (* Mobile Phone*) {
    state DeInitialized {
        when GUI event GUI_REQ_INIT(BSid:integer, BSPinCode:integer)
            (* User requests init with specific BS id and PIN code *)
            actions
                L2CAP request connection to id (BSid) with
                protocol("CLT") returning (channel ch)
            transfer to Initializing(ch,BSPinCode)
    }
    state Initializing {
        state WaitingForChannel(BSch:channel, BSPinCode:integer) {
            when L2CAP connection response from (BSch) is (false)
                (* No such BS found *)
                transfer to DeInitialized
            when L2CAP connection response from (BSch) is (true)
                (* Channel established to BS *)
                transfer to WaitingForBSInitReply(BSch)
        state WaitingForBSInitReply(BSch:channel) {
            ...
```

Figure 12: Part of a program in ExSpecState.

5.2 Integrating the diagram generation

The reuse of the front-end and back-end modules currently relies on that all programming languages use the names `StateDecl` and `TransitionDecl` for the non-terminals modelling states and transitions.

In the glue module for ExSpecState, definitions for the attributes `sourceState` and `targetState` in a `TransitionDecl` node are added. The attribute `targetState` could be defined using the lookup facilities provided by the name analysis of ExSpecState (as was done in the glue module of TinyState). In fact it doesn't need to be looked up since each `Use` node is already equipped with an attribute (`decl`) referencing its corresponding declaration. The `sourceState` is in ExSpecState a reference to the declaration of the state in which the transition is declared i.e. we have to look for the closest ancestor node of type `StateDecl` in the AST. The principal part of the glue module for ExSpecState is shown in Fig. 13 with equations for `sourceState`, `targetState`, `stateLabel`, `transitionLabel` and `ASTroot`. The attribute `parentState` used in the definitions is declared in `StateDecl` nodes and defined to reference the closest ancestor of type `StateDecl` in the AST. If there is no such ancestor it references the ancestor of type `Process` instead. Labels for states are in principle defined by appending the labels of its parent states (the symbol "&" is used for appending). If a process named `P` contains a declaration of a state named `S1` which in turn contains a declaration of a state named `S2` then the label of `S2` will be `P_S1_S2`. For transitions the label is the comment attached to its

declaration if present otherwise the name of the event causing the transition.

Non-terminal	*Attributes*	*Equations and functions*
TransitionDecl		eq sourceState := parentState eq targetState := inspect $X := a_Use.decl when StateDecl do $X otherwise none eq transitionLabel := inspect $X := a_OptCommentList when CommentList do $X.commentText otherwise a_Event_Decl.eventName
StateDecl		eq stateLabel := inspext $X := parentState when StateDecl do $X.stateLabel&"_"&a_ID.val otherwise processName&"_"&a_ID.val
ANYNODE		eq ASTroot := root;

Figure 13: Part of the glue module for ExSpecState.

6 Related work

Our visualization technique can be characterized as follows:

- *static code visualization:* the scope of our technique is restricted to static code visualizations, i.e., visualizations that can be derived from the program code (in contrast to dynamic visualizations like execution visualization and algorithm animation).
- *open system:* visualizations are not built into the environment, but can be added as desired.
- *declarative specifications:* visualizations are specified using a declarative formalism, rather than explicitly programmed.
- *language independent:* the visualizations can be specified independently of the programming language used, and reused for different programming languages by specifying different glue modules
- *visualization tool independent:* different visualization tools can be used by specifying different back-end modules

There are many systems that have support for some kind of program visualization. However, most systems are not open, but provide support only for a set of built-in visualizations. They are usually also language dependent and provide support only for a predefined set of programming languages. One example is Panorama, a visual environment for Java/C/C++ which supports visualizations like call graphs and flow diagrams [2]. Other examples include Rational Rose [3] and TogetherJ [4]. These latter systems provide support not only for program visualization, but also for visual programming, i.e., the possibility to edit the diagrams. They support round-trip engineering where the user can edit diagrams with auto-updating source code and also edit source code with auto-updating diagrams. Dedicated visual programming environments include Prograph [8]. Language-based approaches to visual programming includes graph-grammar based environments, such as Progres [23]. A fundamental difference between these tools and ours is that they use graphs as the main program representation, whereas in our approach the main representation is an abstract syntax tree described by a context-free grammar.

Pavane [21] is a tool that, like ours, takes a declarative and language-independent approach to visualization. However, the scope of Pavane is algorithm animation rather than static code visualization. The animator defines a mapping from program states to graphical objects. For some languages, like C++, some annotation of the program code is needed.

In [22], another language-based approach for using attribute grammars for visualization is described. However, this approach is restricted to tree-structured visualizations of syntax trees. The system integrates the language-based editor generator CENTAUR with a visualization tool FIGUE which is capable of displaying trees specified as Lisp lists. An example of its use is in visualizing mathematical formulas in their standard mathematical form. Attribute grammars are used, but only in the integration process of the tools. A given visualization can be reused for all languages specified in CENTAUR but the only structural aspect of a program that can be visualized is its abstract syntax tree. The solution seems to closely couple the tools with no intention of possible exchange of the visualization tool.

A language independent program visualization technique is described in [9]. Control structure diagrams (CSD) are generated automatically from source code. CSD diagrams add some graphical notations to pretty-printed source code in order to depict control structures and levels of nesting. The tool works in two phases. During the first phase markup tags are inserted in the source code to identify all control structures. In the second phase the tags are used to render the visualization. The renderer is completely language independent but a new tagger must be developed for every language. The separation of a language dependent phase from a language independent one resembles our modularization technique. The scope of the visualization is restricted to CSD diagrams, and diagrams with arbitrary relationships between program structures can thus not be handled.

7 Conclusions and future work

We have described a technique to integrate visualizations in language-based environments and how they can be specified declaratively in RAGs. We have also shown how the solution can be modularized to facilitate reuse for different programming languages and exchange of the external drawing tool.

Attribute grammars allow specification of context-sensitive aspects of a language such as semantic checking and code generation. The specifications are declarative and thus potentially clearer and more concise than imperative code since they only state facts about the final computation results and not the order of computation. The extension of canonical attribute grammars to RAGs makes it easy to define grammar aspects where non-local dependencies play an important role. An example is the visualization aspects as shown in sections 3 and 4. Reference attributes permit a clear and concise way of describing the non-local dependencies in the AST that constitute the graph on which the visualization is based. Information can be propagated along the reference attributes describing the graph structure thus facilitating the generation of a correct representation for an external drawing tool.

For the definition of an individual attribute, one can always argue if an imperative procedure or a declarative function is easiest to understand. This was touched upon in section 4.2 where different techniques for handling cycles in the graphs were discussed. In principle, it would be possible to allow imperative definition of an attribute, provided this code does not produce any net side effects (i.e., side effects that remain after execution of the code). To support such imperative specification in a safe way could be a topic of future work.

It is straightforward to express a solution in general terms in APPLAB. Rather than using information about the structure of the syntax tree for a certain language, a more general approach can be taken by representing the important structures using reference attributes. This allows the front-end and back-end of the specification to be reused for different programming languages.

The APPLAB specification language currently supports modularization by allowing attribute definitions for a certain aspect of the grammar to be textually separated from other grammar aspects of the language being specified. As mentioned in section 2.3, a generalization of the module system is needed to make the front-end of our solution completely reusable for all programming languages. Currently, the essence of the front-end is reusable but relies on grammars to use the same names for its node classes. We plan to generalize this by extending the OOSL formalism with a possibility to declare syntactic part objects, similar to part objects in BETA [18] or anonymous inner classes in Java. The back-end is concerned solely with the generation of a proper representation for an external tool. Using different tools for different kinds of visualizations can thereby be achieved by exchanging this module.

In the near future, we also plan to improve our tool integration so that visualizations in external drawing tools can be opened and updated more conveniently; in the current solution the user has to print an attribute to an explicit text file and start the drawing tool by a shell command.

A more long-term challenge is to try to extend the technique so that external visualization tools that support editing, like daVinci, can be used for actually editing the visualized program, and propagate those edits back to the original syntax tree. Preferably, the external tool should include an event propagation mechanism so that each individual editing step could be propagated back to APPLAB. A main challenge in making such integration work is to devise a mechanism that allows the change of a reference attribute value to induce a corresponding change to the AST. For example, changing an edge in the visualization graph means changing the value of reference attributes in the AST. The proper change of the AST to make it consistent must then be found.

Acknowledgements

We are grateful to the anonymous reviewers for constructive comments and to Mathias Haage for providing us with pointers to various work on visualization. This work was partly supported by NUTEK, the Swedish National Board for Industrial and Technical Development.

Bibliography

[1] daVinciv2.1 on-line Documentation, 1998. Available at http://www.tzi.de/~davinci/docs/overviewF.html.

[2] Panorama. A visual environment for Java/C/C++ software testing, quality assurance, documentation and maintenance, 1999. Available at http://www.softwareautomation.com.

[3] Rational Rose. A visual modelling tool, 1999. Available at http://www.rational.com/products/rose/index.jtmpl.

[4] TogetherJ., 1999. Available at http://www.togethersoft.com.

[5] E. Bjarnason. APPLAB User's Guide. Technical report, Dept of Computer Science, Lund University, Sweden, 1995.

[6] E. Bjarnason. Interactive Tool Support for Domain-Specific Languages, December 1997. Licentiate Thesis. Lund University, Sweden.

[7] E. Bjarnason, G. Hedin, and K. Nilsson. Interactive Language Development for Embedded Systems. *Nordic Journal of Computing*, 6(1):36–55, 1999.

[8] P. T. Cox, F. R. Giles, and T. Pietrzykowski. *Prograph. In Visual Object-Oriented Programming*. Manning Publications Co., 1995.

[9] J. H. Cross and T. D. Hendrix. *Language Independent Program Visualization. In Software Visualization. Series on Software Engineering and Knowledge Engineering. Vol 7*. Word Scientific, 1996.

[10] S. Gestegård. Emulation Software for Executable Specifications, 1999. Master's Thesis. LU-CS-EX:99-6.

[11] W. Harrison and H. Ossher. Subject-Oriented Programming – a Critique of Pure Objects. In *Proceedings of 1993 Conference on Object-oriented Programming Systems, Languages and Applications*, 1993.

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

[13] G. Hedin. *Incremental Semantic Analysis*. Ph.D. thesis, Lund University, Lund, Sweden, 1992. LUTEDX/(TECS-1003)/1-276/(1992).

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

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

[16] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. An Algebra for Program Fragments. In *ACM SIGPLAN'85 Symposium on Programming Languages and Programming Environments*, Seattle, Washington, 1985.

[17] O. L. Madsen. On defining semantics by means of extended attribute grammars. In *Semantics-Directed Compiler Generation, Proceedings of a Workshop*, volume 94 of *LNCS*, pages 259–299. Springer, 1980.

[18] O. L. Madsen and B. Møller-Pedersen. Part objects and their location. In *Proceedings of the seventh international conference on Technology of object-oriented languages and systems*, pages 283–297. Prentice Hall International (UK) Ltd., 1992.

[19] E. Magnusson. State Diagram Generation using Reference Attributed grammars. LU-CS-TR:2000-219. Technical report, Dept. of Computer Science, Lund University, Sweden, 2000.

[20] T. Reps and T. Teitelbaum. The synthesizer generator. In *ACM SIGSOFT-/SIGPLAN Symp. on Practical Software Development Environments*, pages 42–48. ACM Press, 1984.

[21] G-C. Roman. *Declarative Visualization. In Software Visualization.* MIT Press, 1998.

[22] C. Roudet. *Visualisation graphique incrémentale par evaluation d'attributes. Stage DEA Informatique de l'essi*. Université Nice, 1994.

[23] A. Schürr. PROGRES, a Visual Language and Environment for PROgramming with Graph REwrite systems. Technical report, RWTH, Aachen, Germany, 1994. AIB 94-11.

PAPER II

JASTADD—AN ASPECT-ORIENTED COMPILER CONSTRUCTION SYSTEM

Abstract

We describe JastAdd, a Java-based system for compiler construction. JastAdd is centered around an object-oriented representation of the abstract syntax tree where reference variables can be used to link together different parts of the tree. JastAdd supports the combination of declarative techniques (using Reference Attributed Grammars) and imperative techniques (using ordinary Java code) in implementing the compiler. The behavior can be modularized into different aspects, e.g. name analysis, type checking, code generation, etc., that are woven together into classes using aspect-oriented programming techniques, providing a safer and more powerful alternative to the Visitor pattern. The JastAdd system is independent of the underlying parsing technology and supports any non-circular dependencies between computations, thereby allowing general multi-pass compilation. The attribute evaluator (optimal recursive evaluation) is implemented very conveniently using Java classes, interfaces, and virtual methods.

Published as: G. Hedin and E. Magnusson, "JastAdd—an aspect-oriented compiler construction system". SCP - Science of Computer Programming, 47(1):37-58. Elsevier. November 2002.

1 Introduction

Many existing parser generators have only rudimentary support for further compilation. Often, the support is limited to simple semantic actions and tree building during parsing. Systems supporting more advanced processing are usually based on dedicated formalisms like attribute grammars and algebraic specifications. These systems often have their own specification language and can be difficult to integrate with handwritten code, in particular when it is desired to take full advantage of state-of-the-art object-oriented languages like Java. In this paper we describe JastAdd, a simple yet flexible system which allows compiler behavior to be implemented conveniently based on an object-oriented abstract syntax tree. The behavior can be modularized into different aspects, e.g., name analysis, type checking, code generation, etc., that are combined into the classes of the abstract syntax tree. This technique is similar to the *introduction* feature of aspect-oriented programming in AspectJ [17]. A common alternative modularization technique is to use the Visitor design pattern [9,24]. However, the aspect-oriented technique has many advantages over the Visitor pattern, including full type checking of method parameters and return values, and the ability to associate not only methods but also fields to classes.

When implementing a compiler, it is often desirable to use a combination of declarative and imperative code, allowing results computed by declarative modules to be accessed by imperative modules and vice versa. For example, an imperative module implementing a print-out of compile-time errors can access the error attributes computed by a declarative module. In JastAdd, imperative code is written in aspect-oriented Java code modules. For declarative code, JastAdd supports Reference Attributed Grammars (RAGs) [14]. This is an extension to attribute grammars that allows attributes to be references to abstract syntax tree nodes, and attributes can be accessed remotely via such references. RAGs allow name analysis to be specified in a simple way also for languages with complex scope mechanisms like inheritance in object-oriented languages. The formalism makes it possible to use the Abstract Syntax Tree (AST) itself as a symbol table, and to establish direct connections between identifier use sites and declaration sites by means of reference attributes. Further behavior, whether declarative or imperative, can be specified easily by making use of such connections. The RAG modules are specified in an extension to Java and are translated to ordinary Java code by the system.

Our current version of the JastAdd system is built on top of the LL parser generator JavaCC [4]. However, its design is not specifically tied to JavaCC: the parser generator is used only to parse the program and to build the abstract syntax tree. The definition of the abstract syntax tree and the behavior modules are completely independent of JavaCC and the system could as well have been based on any other parser generator for Java such as the LALR-based system CUP [2] or the LL-based system ANTLR [1].

The JavaCC system includes tree building support by means of a preprocessor called JJTree. JJTree allows easy specification of what AST nodes to generate during parsing, and also supports automatic generation of AST classes. However, there is no mechanism in JJTree to update AST classes once they have been generated, so if the AST classes need more functionality than is generated, it is up to the programmer to modify the generated classes by hand and to update the classes after changes in the grammar. In JastAdd, this tedious and error-prone procedure is completely avoided by allowing handwritten and generated code to be kept in separate modules. JastAdd uses the JJTree facility for annotating the parser specification with tree-building actions, but the AST classes are generated directly by JastAdd, rather than relying on the JJTree facility for this. SableCC [11] and JTB [3] are other Java-based systems that have a similar distinction between generated and handwritten modules. While both SableCC and JTB support the Visitor pattern for adding behavior, neither one supports aspect-oriented programming nor declarative specification of behavior like attribute grammars.

The attribute evaluator used in JastAdd is an optimal recursive evaluator that can handle arbitrary acyclic attribute dependencies. If the dependencies contain cycles, these are detected at attribute evaluation time. The evaluation technique is in principle the same as the one used by many earlier systems such as Madsen [21], Jalili [15], and Jourdan [16]: an access to an attribute value is replaced by a function call which computes the appropriate semantic function for the value and then caches the computed value for future accesses to the same attribute. A cache flag is used to keep track of whether the value has been computed before and is cached. A cycle flag is used to keep track of attributes involved in an evaluation so that cyclic dependencies can be detected at evaluation time. While these earlier systems used this evaluation algorithm for traditional attribute grammars, it turns out that this algorithm is also applicable to RAGs [14]. Our implementation in JastAdd differs from earlier implementations in its use of object-oriented programming for convenient coding of the algorithm.

The rest of the paper is outlined as follows. Section 2 describes the object-oriented ASTs used in JastAdd. Section 3 describes how imperative code can be modularized according to different aspects of compilation and woven together into complete classes. Section 4 describes how RAGs can be used in JastAdd and Section 5 how they are translated to Java. Section 6 discusses related work and Section 7 concludes the paper.

2 Object-oriented abstract syntax trees

2.1 Connection between abstract and parsing grammars

The basis for specification in JastAdd is an abstract context-free grammar. An abstract grammar describes the programs of a language as typed trees rather than as strings. Usually, an abstract grammar is essentially a simplification of a parsing

grammar, leaving out the extra nonterminals and productions that resolve parsing ambiguities (e.g., terms and factors) and leaving out tokens that do not carry semantic values. In addition, it is often useful to have fairly different structure in the abstract and parsing grammars for certain language constructs. For example, expressions can be conveniently expressed using EBNF rules in the parser, but are more adequately described as binary trees in the abstract grammar. Also, parsing-specific grammar transformations like left factorization and elimination of left recursion for LL parsers are undesirable in the abstract grammar.

Most parsing systems that support ASTs make use of various automatic rules and annotations in order to support abstraction of the parsing grammar. In JastAdd, the abstract grammar is independent of the underlying parsing system. The parser is simply a front end whose responsibility it is to produce abstract syntax trees that follow the abstract grammar specification.

2.2 Object-oriented abstract grammar

When using an object-oriented language like Java, the most natural way of representing an AST is to model the language constructs as a class hierarchy with general abstract classes like Statement and Expression, and specialized concrete classes like Assignment and AddExpression. Methods and fields can then be attached to the classes in order to implement compilation or interpretation. This design pattern is obvious to any experienced programmer, and documented as the *Interpreter* pattern in [9].

Essentially, this object-oriented implementation of ASTs can be achieved by viewing nonterminals as abstract superclasses and productions as concrete subclasses. However, this two-level hierarchy is usually insufficient from the modelling point of view where it is desirable to make use of more levels in the class hierarchy. For this reason, JastAdd makes use of an explicit object-oriented notation for the abstract grammar, similar to [13], rather than the usual nonterminal/production-based notation. This allows nonterminals with a single production to be modelled by a single class. It also allows additional superclasses to be added that would have no representation in a normal nonterminal/production grammar, but are useful for factoring out common behavior or common subcomponents. Such additional superclasses would be unnatural to derive from a parsing grammar, which is yet another reason for supplying a separate specification of the abstract grammar.

The abstract grammar is a class hierarchy augmented with subcomponent information corresponding to production right-hand sides. For example, a class Assignment typically has two subcomponents: an Identifier and an Expression. Depending on what kind of subcomponents a class has, it is categorized as one of the following typical kinds (similar to many other systems):

list The class has a list of components of the same type.

optional The class has a single component which is optional.

Tiny.ast

```
 1  Program ::= Block;
 2  Block ::= Decl Stmt;
 3  abstract Stmt;
 4  BlockStmt : Stmt ::= Block;
 5  IfStmt : Stmt ::= Exp Stmt OptStmt;
 6  OptStmt ::= [Stmt];
 7  AssignStmt : Stmt ::= IdUse Exp;
 8  CompoundStmt : Stmt ::= Stmt*;
 9  abstract Decl;
10  BoolDecl: Decl ::= <ID>;
11  IntDecl : Decl ::= <ID>;
12  abstract Exp;
13  IdUse : Exp ::= <ID>;
14  Add : Exp ::= Exp Exp;
15  ...
```

Figure 1: Abstract grammar for Tiny

token The class has a semantic value extracted from a token.

aggregate The class has a set of components which can be of different types.

The subcomponent information is used for generating suitable access methods that allow type safe access to methods and fields of subcomponents.

2.3 An example: Tiny

We will use a small toy block-structured language, *Tiny*, as a running example throughout this paper. Blocks in Tiny consist of a single variable declaration and a single statement. A statement can be a compound statement, an if statement, an assignment statement, or a new block.

Figure 1 shows the object-oriented abstract grammar for Tiny. (The line numbers are not part of the actual specification.) All the different kinds of classes are exemplified: An aggregate class `IfStmt` (line 5), a list class `CompoundStmt` (line 8), an optional class `OptStmt` (line 6), and a token class `BoolDecl` (line 10). The classes are ordered in a single-inheritance class hierarchy. For example, `BlockStmt`, `IfStmt`, `AssignStmt`, and `CompoundStmt` (lines 4, 5, 7, and 8) are all subclasses to the abstract superclass `Stmt` (line 3).

From this abstract grammar, the JastAdd system generates a set of Java classes with access methods to their subcomponents. Figure 2 shows some of the generated classes to exemplify the different kinds of access interfaces to different kinds of classes. Note that for an aggregate class with more than one subcomponent of the same type, the components are automatically numbered, as for the class `ASTAdd`.

```
abstract class ASTStmt {
}
class ASTIfStmt extends ASTStmt {
   ASTExp getExp() { ... }
   ASTStmt getStmt() { ... }
   ASTOptStmt getOptStmt() { ... }
}
class ASTOptStmt {
   boolean hasStmt() { ... }
   ASTStmt getStmt() { ... }
}
class ASTCompoundStmt extends ASTStmt {
   int getNumStmt() { ... }
   ASTStmt getStmt(int k) { ... }
}
class ASTBoolDecl extends ASTDecl {
   String getID() { ... }
}
class ASTAdd extends ASTExp {
   ASTExp getExp1() { ... }
   ASTExp getExp2() { ... }
}
```

Figure 2: Access interface for some of the generated AST classes

Behavior can be added to the generated classes in separate aspect-oriented modules. Imperative behavior is added in `Jadd` modules that contain methods and fields as described in Section 3. Declarative behavior is added in `Jrag` modules that contain equations and attributes as described in Section 4. Figure 3 shows the Jastadd system architecture. The `jadd` tool generates AST classes from the abstract grammar and weaves in the imperative behavior defined in `Jadd` modules. The `jrag` tool translates the declarative `Jrag` modules into an imperative `Jadd` module, forming one of the inputs to the `jadd` tool. This translation is described in more detail in Section 5.

2.4 Superclasses and interfaces

When adding behavior it is often found that certain behavior is relevant for several classes although the classes are unrelated from a parsing point of view. For example, both `Stmt` and `Exp` nodes may have use for an `env` attribute that models the environment of visible identifiers. In Java, such sharing of behavior can be supported either by letting the involved classes inherit from a common superclass or by letting them implement a common interface. JastAdd supports both ways. Common superclasses are specified in the abstract grammar. Typically, it is useful to introduce a superclass `Any` that is the superclass of all other AST classes. For the example in Figure 1, this would be done by adding a new class "`abstract Any;`" into the abstract grammar and adding it as a superclass to all other classes

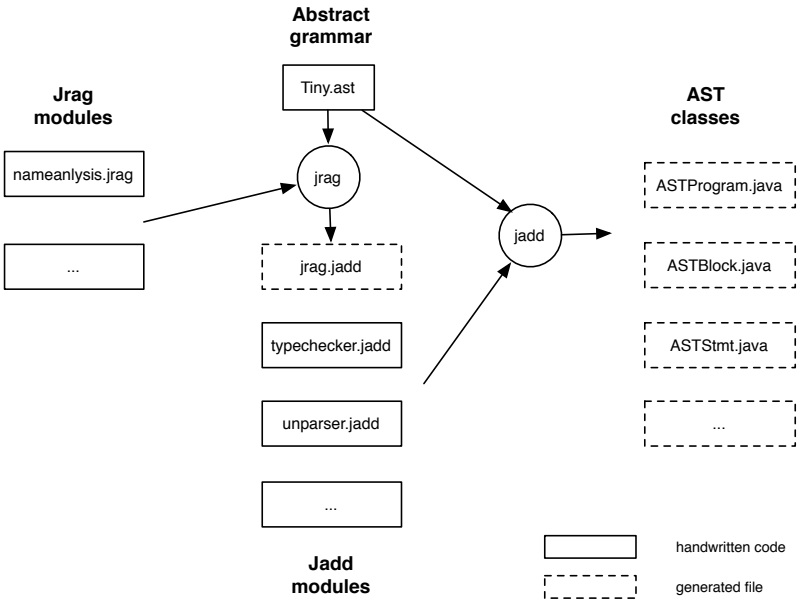

Figure 3: Architecture of the JastAdd system

that do not already have a superclass. Figure 4 shows the corresponding class diagram.

Such common superclasses allows common default behavior to be specified and to be overridden in suitable subclasses. For example, default behavior for all nodes might be to declare an attribute `env` and to by default copy the `env` value from each node to its components by adding an equation to `Any`. AST classes that introduce new scopes, e.g. `Block`, can then override this behavior by supplying a different equation.

Java interfaces are more restricted in that they can include only method interfaces and no fields or default implementations. On the other hand, they are also more flexible, allowing, e.g., selected AST classes to share a specific interface orthogonally to the class hierarchy. Such selected interface implementation is specified as desired in the behavior modules and will be discussed in Section 3.4.

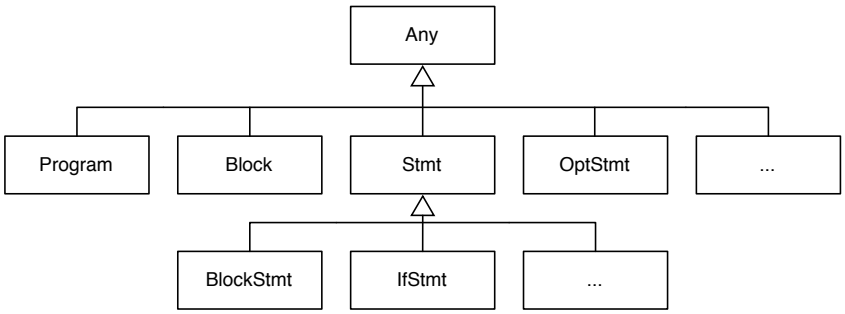

Figure 4: Class diagram after adding the superclass `Any`

2.5 Connection to the parser generator

Building the tree

JastAdd relies on an underlying parsing system for parsing and tree-building. The abstract grammar is not tied to any specific parsing grammar or parsing algorithm and there is thus normally a gap between these grammars that must be bridged. To aid the compiler writer, the JastAdd system generates a method `syntaxCheck()` which can be called to check that the built tree actually follows the abstract grammar.

Currently, JastAdd uses JavaCC/JJTree as its underlying parsing and tree-building system. JJTree allows easy specification of what AST nodes to generate during parsing. A stack is used to give the programmer control over the order in which to insert the individual nodes, so that the structure of the constructed AST

does not have to match the structure of the parse. For example, expressions that are parsed as a list can easily be built as a binary AST. In this way, JJTree allows the gap between the parsing and abstract grammars to be bridged fairly easily.

Token semantic values

When building the AST, information about the semantic values of tokens needs to be included. To support this, JastAdd generates a set-method as well as a get-method for each token class. For example, for the token class `BoolDecl` in Figure 1, a method `void setID(String s)` is generated. This method can be called as an action during parsing in order to transmit the semantic value to the AST.

3 Adding imperative behavior

Object-oriented languages lend themselves very nicely to the implementation of compilers. It is natural to model an abstract syntax tree using a class hierarchy where nonterminals are modelled as abstract superclasses and productions as specialized concrete subclasses, as discussed in Section 2. Behavior can be implemented easily by introducing abstract methods on nonterminal classes and implementing them in subclasses. However, a problem is that to make use of the object-oriented mechanisms, the class hierarchy imposes a modularization based on language constructs whereas the compiler writer also wants to modularize based on aspects in the compiler, such as name analysis, type checking, error reporting, code generation, and so on. Each AST class needs to include the code related to all of the aspects and in traditional object-oriented languages it is not possible to provide a separate module for each of the aspects. This is a classical problem that has been discussed since the origins of object-oriented programming.

3.1 The Visitor pattern

The Visitor design pattern is one (partial) solution to this problem [9]. It allows a given method that is common to all AST nodes to be factored out into a helper class called a Visitor containing an abstract `visit(C)` method for each AST class `C`. To support this programming technique, all AST classes are equipped with a generic method `accept(Visitor)` which delegates to the appropriate `visit(C)` method in the Visitor object. For example, a `Visitor` subclass `TypeCheckingVisitor` can implement type checking in its `visit` methods. Type checking of a program is started by calling `accept` on the root node with the `TypeCheckingVisitor` as a parameter.

There are several limitations to the Visitor pattern, however. One is that only methods can be factored out; fields must still be declared directly in the classes, or be handled by a separate mechanism. For example, in type checking it is useful to

associate a field `type` with each applied identifier, and this cannot be handled by the Visitor pattern. Another drawback of the Visitor pattern is that the parameter and return types can not be tailored to the different visitors – they must all share the same interface for the visit methods. For example, for type checking expressions, a desired interface could be

```
Type typecheck(Type expectedType)
```

where `expectedType` contains the type expected from the context and the `typecheck` method returns the actual type of the expression. Using the Visitor pattern, this would have to be modelled into visit methods

```
Object visit(C node, Object arg)
```

to conform to the generic visit method interface.

3.2 Aspect-oriented programming

A more powerful alternative to the Visitor pattern is to introduce an explicit modularization mechanism for aspects. This is the approach used in JastAdd. Our technique is similar to the *introduction* feature of the aspect-oriented programming system AspectJ [17].

For each aspect, the appropriate fields and methods for the AST classes are written in a separate file, a *Jadd module*. The JastAdd system is a class weaver: it reads all the Jadd modules and weaves the fields and methods into the appropriate classes during the generation of the AST classes. This approach does currently not support separate compilation of individual Jadd modules, but, on the other hand, it allows a suitable modularization of the code and does not have the limitations of the Visitor pattern.

The Jadd modules use normal Java syntax. Each module simply consists of a list of class declarations. For each class matching one of the AST classes, the corresponding fields and methods are inserted into the generated AST class. It is not necessary to state the superclass of the classes since that information is supplied by the abstract grammar. Figure 5 shows an example. The `typechecker.jadd` module performs type checking for expressions and computes the boolean field `typeError`. The `unparser.jadd` module implements an unparser which makes use of the field `typeError` to report type-checking errors.

The Jadd modules may use fields and methods in each other. This is illustrated by the unparser module which uses the `typeError` field computed by the type checking module. The Jadd modules may freely use other Java classes. This is illustrated by the unparsing module which imports a class `Display`. The import clause is transmitted to all the generated AST classes. Note also that the Jadd modules use the generated AST access interface described in Section 2. An example of a complete AST class generated by the JastAdd system is shown in Figure 6.

typechecker.jadd

```
...
class IfStmt {
  void typeCheck() {
    getExp().typeCheck("Boolean");
    getStmt().typeCheck();
    getOptStmt().typeCheck();
  }
}
class Exp {
  abstract void typeCheck(String expectedType);
}
class Add {
  boolean typeError;
  void typeCheck(String expectedType) {
    getExp1().typeCheck("int");
    getExp2().typeCheck("int");
    typeError = expectedType != "int";
  }
}
...
```

unparser.jadd

```
import Display;
class Stmt {
  abstract void unparse (Display d);
}
class Exp {
  abstract void unparse (Display d);
}
class Add {
  void unparse (Display d) {
    ...
    if (typeError)
      d.showError("type mismatch");
  }
}
...
```

Figure 5: Jadd modules for typechecking and unparsing.

ASTAdd.java

```
class ASTAdd extends ASTExp {
  // Access interface
  ASTExp getExp1() { ... }
  ASTExp getExp2() { ...}

  // From typechecker.jadd
  boolean typeError;
  void typeCheck(String expectedType) {
    getExp1().typeCheck("int");
    getExp2().typeCheck("int");
    typeError = expectedType != "int";
  }

  // From unparser.jadd
  void unparse(Display d) {
    ...
    if (typeError)
      d.showError("type mismatch");
     ...
  }
}
```

Figure 6: Woven complete AST class

In the current JastAdd system, the names of the generated classes are by default prefixed by the string "AST" as in the JavaCC/JJTree system.

3.3 Using the AST as a symbol table

In traditional compiler writing it is common to build symbol tables as large data structures, separate from the parse tree. The use of object-oriented ASTs makes it convenient to use another approach where the AST itself is used as a symbol table, connecting each AST node that serves as an applied identifier to the corresponding AST node that serves as the declaration. This technique is particularly powerful in combination with aspect-oriented programming. Each part of the compiler that computes a certain part of the "symbol table" can be separated into a specific aspect, imperative or declarative.

Consider the language Tiny in Figure 1. Name analysis involves connecting each applied identifier (`IdUse` node) to its corresponding declared identifier (`Decl` node). For example, taking an imperative approach, this can be implemented by declaring a field `Decl myDecl` in class `IdUse` and by writing methods that traverse the AST and set each such field to the appropriate `Decl` node. Typically, this computation will make use of some efficient representation of the declarative environment, e.g. a hash table of references to the visible `Decl` nodes. But once the `myDecl` fields are computed, the hash table is no longer needed.

Other aspects can add fields and methods to the `Decl` nodes and access that information from the `IdUse` nodes via the `myDecl` field. For example, a type analysis aspect can add a type field to each `Decl` node and access that field from each `IdUse` node during type checking. A code generation aspect can add a field for the activation record offset to each `Decl` node and access that field from each `IdUse` node for generating code.

More complex type information such as structured and recursive types, class hierarchies, etc. is available more or less directly through the `myDecl` fields. For example, a class declaration node will contain a subnode that is an applied identifier referring to the superclass declaration node. More direct access to the superclass can easily be added as an extra field or method of the class declaration nodes. In this way, once the `myDecl` fields are computed, the AST itself serves as the symboltable.

The different compiler aspects can be implemented as either imperative or declarative aspect modules. Section 4 describes how to implement the name analysis declaratively, defining `myDecl` as a synthesized attribute rather than as a field and specifying its value using equations rather than computing it with imperative methods.

3.4 Adding interface implementations to classes

As mentioned in Section 2.4, aspect modules may add interface implementations to the AST classes. One use of this is to relate AST classes that are syntactically unrelated. As an example, consider implementing name analysis for a language which has many different block-like constructs, e.g. class, method, compound-statement, etc. Each of these block-like constructs should have a method `lookup` which looks up a name among its local declarations, and if not found there, delegates the call to some outer block-like construct. This can be implemented in a name analysis aspect by introducing an interface `Env` with the abstract method `lookup` and adding this interface implementation to each of the involved AST classes.

Another use of interfaces is to relate AST classes to other externally defined classes. One use of this is in order to apply the *Null pattern* for references within the AST. The Null pattern recommends that null references are replaced by references to real (but usually empty) objects, thereby removing the need for specific handling of null references in the code [25]. For example, in the case of an undeclared identifier, the `myDecl` field could refer to a special object of type `NotDeclared`, rather than being null. This can be implemented in a name analysis aspect by introducing an interface `Declaration` whose implementation is added both to the class `NotDeclared` and to the involved AST classes. Naturally, the type of `myDecl` should in this case be changed to `Declaration` as well.

3.5 Combining visitors with aspect-oriented programming

Visitors have serious limitations compared to aspect-oriented programming as discussed earlier. They support modularization only of methods and not of fields, and they do not support type-checking of the method arguments and return values. However, there are certain applications where visitors actually may be slightly simpler to use than Jadd modules, namely when the computation can be formulated as a regular traversal and when the untyped method arguments can be replaced by typed visitor instance variables. This is illustrated in Figure 7 where the visitor implementation is slightly simpler than the corresponding Jadd module. In the visitor implementation, the traversal method has been factored out into a superclass `DefaultTraversingVisitor` which can be reused for other visitors. Furthermore, the ErrorCollector object which is used by all visit methods is declared directly in the visitor, rather than supplied as an argument as in the Jadd module.

Visitors and aspect-oriented programming can be freely combined so that each subproblem is solved by the most suitable implementation technique. For example, the `visit(IdUse)` method in the visitor in Figure 7 accesses the field `myDecl` that can be supplied by a Jadd (or Jrag) module.

JastAdd stays backward compatible with JavaCC/JJTree by generating the same visitor support as JJTree (the same `"accept"` methods), thereby allowing existing JJTree projects to be more easily migrated to JastAdd. The visitor support has also been useful for bootstrapping the JastAdd system.

4 Adding declarative behavior

In addition to imperative modules it is valuable to be able to state computations declaratively, both in order to achieve a clearer specification and to avoid explicit ordering of the computations, thereby avoiding a source of errors that are often difficult to debug.

JastAdd supports the declarative formalism Reference Attributed Grammars (RAGs) which fits nicely with object-oriented ASTs. In attribute grammars, computations are defined declaratively by means of attributes and equations. Each attribute is defined by an equation and can be either *synthesized* (for propagating information upwards in the AST) or *inherited* (for propagating information downwards in the AST). An equation defines either a synthesized attribute in the same object, or an inherited attribute in a child object. An attribute can be thought of as a read-only field whose value is equal to the right-hand side of its defining equation.

The important extension in RAGs (as compared to traditional attribute grammars) is the support for reference attributes. The value of such an attribute is a reference to an object. In particular, a node q can contain a reference attribute referring to another node r, arbitrarily far away from q in the AST. This way arbitrary connections between nodes can be established, and equations in q can access

visitor – ErrorChecker.java

```
class ErrorChecker extends DefaultTraversingVisitor {
  ErrorCollector errs = new ErrorCollector();

  void visit(IdUse node) {
    if (node.myDecl==null) errs.add(node, "Missing declaration");
  }

  void visit(...
}
```

Jadd module – errorchecker.jadd

```
class Any {
  void errorCheck(ErrorCollector errs) {
    for (int k=0;k<getNumChildren();k++)
      getChild(k).errorCheck(errs);
  }
}

class IdUse {
  void errorCheck(ErrorCollector errs) {
    if (myDecl==null) errs.add(this, "Missing declaration");
  }
}

class ...
```

Figure 7: Two alternative implementations of error checking

attributes in *r* via the reference attribute. Typically, this is used for connecting applied identifiers to their declarations.

In a Java-based RAG system, the type of a reference attribute can be either a class or an interface. The interface mechanism gives a high degree of flexibility. For example, to implement name analysis, the environment of visible declarations can be represented by a reference attribute `env` of an interface type `Env`. Each language construct that introduces a new declarative environment, e.g., `Block`, `Method`, `Class`, and so on, can implement the `Env` interface, providing a suitable implementation of a function `lookup` for looking up declarations.

RAGs are specified in separate files called *Jrag* modules. The Jrag language is a slightly extended and modified version of Java. A Jrag module consists of a list of class declarations, but instead of fields and methods, each class contains attributes and equations. Ordinary methods may be declared as well and used in the equations. However, in order to preserve the declarative semantics of attribute grammars, these methods should in effect be functions, containing no side effects that are visible outside the method.

The syntax for attributes and equations is similar to Java. Attribute declarations are written like field declarations, but with an additional modifier `"syn"` or `"inh"` to indicate if the attribute is synthesized or inherited. Java method call syntax is used for accessing attributes, e.g., `a()` means access the value of the attribute `a`. Equations are written like Java assignment statements. Equations for synthesized attributes can be written directly as part of the attribute declaration (using the syntax of variable initialization in Java). For access to components, the generated access methods for ASTs is used, e.g., `getStmt()` for accessing the `Stmt` component of a node.

Jrag modules are aspect-oriented in a similar way as Jadd modules: they add attributes and equations to AST classes analogously to how Jadd modules add fields and methods. The JastAdd system translates the Jrag modules to Java and combines them into a Jadd module before weaving. This translation is described in Section 5.

4.1 An example: name analysis and type checking

Figure 8 shows an example of a Jrag module for name analysis of the language Tiny. (Line numbers are not part of the actual specification.) All blocks, statements, and expressions have an inherited attribute `env` representing the environment of visible declarations. The `env` attribute is a reference to the closest enclosing `Block` node, except for the outermost `Block` node whose `env` is `null`, see the equations on lines 2 and 6. All other `env` definitions are trivial copy equations, e.g., on lines 22 and 23.

The goal of the name analysis is to define a connection from each `IdUse` node to the appropriate `Decl` node (or to `null` if there is no such declaration). This is done by a synthesized reference attribute `myDecl` declared and defined at line

nameanalysis.jrag

```
1  class Program {
2    getBlock().env = null;
3  }
4  class Block {
5    inh Block env;
6    getStmt().env = this;
7    ASTDecl lookup(String name) {
8      return
9         (getDecl().name().equals(name))
10        ? getDecl()
11        : (env() == null) ? null
12        : env().lookup(name);
13   }
14 }
15 class Stmt {
16   inh Block env;
17 }
18 class BlockStmt {
19   getBlock().env = env();
20 }
21 class AssignStmt {
22   getIdUse().env = env();
23   getExp().env = env();
24 }
25 class Decl {
26   syn String name;
27 }
28 class Exp {
29   inh Block env;
30 }
31 class Add {
32   getExp1().env = env();
33   getExp2().env = env();
34 }
35 class IdUse {
36   inh Block env;
37   syn Decl myDecl= env().lookup(name());
38   syn String name = getID();
39 }
40 class IntDecl {
41   name = getID();
42 }
43 class BoolDecl {
44   name = getID();
45 }
```

Figure 8: A Jrag module for name analysis.

typechecker.jrag

```
 1  class Decl     { syn String type; }
 2  class BoolDecl { type = "boolean"; };
 3  class IntDecl  { type = "int"; };
 4  class Exp { syn String type; };
 5  class IdUse {
 6    type = (myDecl()==null)
 7        ? null : myDecl().type()
 8  };
 9  class Stmt { syn boolean typeError; };
10  class AssignStmt {
11    typeError = !getIdUse().type().equals(getExp().type());
12  };
13  ...
```

Figure 9: A Jrag module for type checking.

37. Usual block structure with name shadowing is implemented by the method `lookup` on `Block` (lines 7–13). It is first checked if the identifier is declared locally, and if not, the enclosing blocks are searched by recursive calls to `lookup`.

The `lookup` method is an ordinary Java method, but has been coded as a function, containing only a return statement and no other imperative code. As an alternative, it is possible to code it imperatively using ordinary if-statements. However, it is good practice to stay with function-oriented code as far as possible, using only a few idioms for simulating, e.g., let-expressions. Arbitrary imperative code can be used as well, but then it is up to the programmer to make sure the code has no externally visible side effects.

Figure 9 shows a type checking module that uses the `myDecl` attribute computed by the name analysis. This is a typical example of how convenient it is to use the AST itself as a symbol table and to extend the elements as needed in separate modules. The type checking module extends `Decl` with a new synthesized attribute `type` (line 1). This new attribute is accessed in `IdUse` in order to define its `type` attribute (lines 6–7). The types of expressions are then used as usual to do type checking as shown for the `AssignStmt` (line 11).

The examples are written to be self-contained and straight-forward to understand. For a realistic language several changes would typically be done. The copy equations for `env` would be factored out into a common superclass `Any`, thereby making the specification substantially more concise. The type for `env` attributes would typically also be generalized. In the example we simply used the class `Block` from the abstract grammar as the type of the `env` attribute. For a more complex language with several different kinds of block-like constructs, an interface `Env` can be introduced to serve as the type for `env`. Each different block-like construct (procedure, class, etc.) can then implement the `Env` interface in a suitable way. The Null pattern could be applied, both for the `env` and the `myDecl` attributes, in order to avoid null tests such as on line 11 in Figure 8 and on line 6

in Figure 9. A more realistic language would also allow several declarations per block, rather than a single one as in Tiny. Typically, each block would be extended with a hash table or some other fast dictionary data type to support fast lookup of declarations. Types would be represented as objects rather than as strings, and the type checker would support better error handling, e.g., not considering the use of undeclared identifiers as type checking errors.

It is illustrative to compare the Jrag type checker in Figure 9 with the imperative one sketched in Figure 5. By not having to code the order of computation the specification becomes much more concise and simpler to read than the imperative type checker.

4.2 Combining declarative and imperative aspects

An important strength of the JastAdd system is the ease with which imperative Jadd aspects and declarative Jrag aspects can be combined. A compiler can be divided into many small subproblems and each be solved declaratively or imperatively depending on which paradigm is most suitable. For example, the name analysis and type analysis can be solved by declarative aspects that define the `myDecl` and `type` attributes. Code generation can be split into a declarative aspect that defines block levels and offsets and an imperative aspect that generates the actual code.

It is always safe for an imperative aspect to use attributes defined in a declarative aspect. Usually, this is the natural way to structure a compiler problem: a core of declarative aspects defines an attribution which is used by a number of imperative aspects to accomplish various tasks such as code generation, unparsing, etc.

In principle, it is also possible to let a declarative aspect use fields computed by an imperative aspect. However, for this to be safe it has to be manually ensured that these fields behave as constants with respect to the declarative aspect, i.e., that the computation of them is completed before any access of them is triggered. For example, it would be possible to write an imperative name analysis module that computes `myDecl` fields and let a declarative type checking module access those fields, provided that the name analysis computation is completed before any other computations start that might trigger accesses from the type checking module.

In some attribute-grammar systems, equations are allowed to call methods in order to trigger desired side-effects, e.g., code generation. This technique is used in systems with evaluation schemes that evaluate all attributes exactly once and where the order of evaluation can be predicted. In JastAdd, this technique is not applicable because of the demand evaluation scheme used which will delay the computation of an attribute until its value is needed. This results in an order of evaluation which is not always possible to predict statically and which does not necessarily evaluate all attributes.

5 Translating declarative modules

The JastAdd system translates Jrag modules to ordinary Java code, weaving together the code of all Jrag modules and producing a Jadd module. Attribute evaluation is implemented simply by realizing all attributes as functions and letting them return the right-hand side of their defining equations, caching the value after it has been computed the first time, and checking for circularities during evaluation. This implementation is particularly convenient in Java where methods, overriding, and interfaces are used for the realization. In the following we show the core parts of the translation, namely how to translate synthesized and inherited attributes and their defining equations for abstract and aggregate AST classes.

5.1 Synthesized attributes

Synthesized attributes correspond exactly to Java methods. A declaration of a synthesized attribute is translated to an abstract method declaration with the same name. For example, recall the declaration of the `type` attribute in class `Decl` of Figure 9

```
class Decl { syn String type; }
```

This attribute declaration is translated to

```
class Decl { abstract String type(); }
```

Equations defining the attribute are translated to implementations of the abstract method. For example, recall the equations defining the `type` attribute in `IntDecl` and `BoolDecl` of Figure 9

```
class IntDecl { type = "int"; }

class BoolDecl { type = "boolean"; }
```

These equations are translated as follows.

```
class IntDecl {
  String type() { return "int"; }
}
class BoolDecl {
  String type() { return "boolean"; }
}
```

5.2 Inherited attributes

An inherited attribute is defined by an equation in the parent node. Suppose a class `X` has an inherited attribute `ia` of type `T`. This is implemented by introducing an interface `ParentOfX` with an abstract method `T X_ia(X)`. Any class which has components of type `X` must implement this interface. If a class has several components of type `X` with different equations for their `ia` attributes, the `X` parameter can be used to determine which equation should be applied in implementing the `X_ia` method. To simplify accesses of the `ia` attribute (e.g. from imperative Jadd modules), a method `T ia()` is added to `X` which simply calls the `X_ia` method of the parent node with itself as the parameter.

For example, recall the declaration of the inherited attribute `env` in class `Stmt` in Figure 8. Both `Block` and `IfStmt` have `Stmt` components and define the `env` attribute of those components:

```
class Stmt {
  inh Block env;
}
class Block {
  getStmt().env = this;
}
class IfStmt {
  getStmt().env = env();
}
```

Since `Stmt` contains declarations of inherited attributes, an interface is generated as follows:

```
interface ParentOfStmt {
  ASTBlock Stmt_env(ASTStmt theStmt);
}
```

The `Block` and `IfStmt` classes must implement this interface. The implementation should evaluate the right-hand side of the appropriate equation and return that value. The translated code looks as follows.

```
class Block implements ParentOfStmt {
  ASTBlock Stmt_env(ASTStmt theStmt) {
    return this;
  }
}
class IfStmt implements ParentOfStmt {
  ASTBlock Stmt_env(ASTStmt theStmt) {
    return env();
  }
}
```

The parameter `theStmt` was not needed in this case, since both these classes have only a single component of type `Stmt`. However, in general, an aggregate class may have more than one component of the same type and equations defining the inherited attributes of those components in different ways. For example, an aggregate class `Example ::= Stmt Stmt` could have the following equations:

```
class Example {
  getStmt1().env = env();
  getStmt2().env = null;
}
```

The translation of `Example` needs to take the parameter into account to handle both equations:

```
class Example implements ParentOfStmt{
  ASTBlock Stmt_env(ASTStmt theStmt) {
    if (theStmt==getStmt1())
      return env();
    else
      return null ;
  }
}
```

Finally, a method `env()` is added to `Stmt` to give access to the attribute value. The method `getParent()` returns a reference to the parent node. The cast is safe since all AST nodes with `Stmt` components must implement the `ParentOfStmt` interface (this is checked by the JastAdd system).

```
class Stmt {
  ASTBlock env() {
    return ((ParentOfStmt) getParent()).Stmt_env(this);
  }
}
```

5.3 Generalizations

The translation described above can be easily generalized to handle lists and optionals. It is also simple to add caching of computed values (to achieve optimal evaluation) and circularity checks (to detect cyclic attribute dependencies and thereby avoid endless recursion) using the same ideas as in other implementations of this algorithm [15, 16, 21].

6 Related work

Recent developments in aspect-oriented programming [10] include the work on AspectJ [17], subject-oriented programming [12], and adaptive programming [20].

AspectJ covers both static aspects through its *introduction* feature and dynamic aspects through its notion of *joinpoints*. The introduction feature allows fields, methods, and interface implementations to be added to classes in separate aspect modules, similar to how our Jadd modules work. Now that a stable release of AspectJ is available and seems to gain wide-spread use it would be attractive to build JastAdd on top of AspectJ rather than using our own mechanism. The focus in AspectJ is, however, on the dynamic aspects rather than the static aspects. The joinpoint model in AspectJ allows code written in aspects to be inserted at dynamically selected execution points. We do not employ such dynamic aspects in JastAdd, but it is a very interesting area of future work to investigate their benefits in compiler construction.

Subject-oriented programming supports static aspects called *subjects* where each subject provides a (possibly incomplete) perspective on a set of classes. There is a strong focus on how to merge subjects that are developed independently. Explicit composition code is used to specify how to merge subjects, allowing, e.g., different subjects to use different names for the same program entity. This approach is powerful, but also more heavy-weight than the technique used in JastAdd.

Adaptive programming focuses on factoring out traversal code and making it robust to structural changes in the class hierarchy. This separation is similar to what can be accomplished by visitors where default traversal strategies can be factored out in superclasses (as in our example in Figure 7). However, adaptive programming goes beyond visitors in several ways. In particular, they do not require the classes involved to be related in a class hierarchy, and they employ generative techniques to generate traversal code from high-level descriptions.

The *fragment system* is a technique for aspect-oriented modularization which predates the above approaches [8, 18]. It provides a general approach to static aspect modularization based on the syntax of the supported language. By using this mechanism for entities in imperative code, dynamic aspect modularization is also supported to a certain extent. The BETA language uses the fragment system as its modularization mechanism.

There are many compiler tools that generate object-oriented ASTs. An early example was the BETA meta programming system (MPS) [22] which also supported aspect modularization to a certain extent via the fragment system mentioned above. However, due to limitations of the separate compilation mechanism it was only possible to factor out methods and not fields.

The Visitor pattern is supported by many recent compiler tools including JJTree [4], SableCC [11], Java Tree Builder [3], and JJForester [19]. These systems

generate AST classes and abstract visitor classes that support various traversal schemes.

There are a few other experimental systems for reference attributed grammars or similar formalisms: the MAX system by Poetzsch-Heffter [23], Boyland's prototype system for the compiler description language APS [6], and our own predecessing system Applab [5]. Similar to JastAdd, these systems stress the modularity with which specifications can be written. In contrast to JastAdd, they all have their own formal languages for specification and do not easily integrate with imperative object-oriented programming in standard languages.

7 Conclusion

We have presented JastAdd, a simple yet flexible and safe system for constructing compilers in Java. Its main features are

- object-oriented ASTs (decoupled from parsing grammars)
- typed access methods for traversing the AST
- aspect modularization for imperative code in the form of fields, methods, and interface implementations
- aspect modularization for declarative code in the form of RAG attributes and equations
- seamless combination of imperative and declarative code

We find this combination very useful for writing practical translators in an easy way. The use of object-oriented ASTs with typed access methods is a natural way of modelling the program. The aspect-modularization is easy to use and makes it easy to change and extend the compiler. We have found it very useful to be able to combine the declarative and imperative techniques for coding a compiler, making it possible to select the most appropriate technique for each individual subproblem. While subsets of these features exist in other systems we are not aware of other systems that combine them all. In particular, we have not found other Java-based compiler tools that are based on aspect-oriented programming or reference attributed grammars.

We have quite substantial experience from using JastAdd in research and education, and also from bootstrapping the system in itself.

Research projects using JastAdd include a Java-to-C compiler and a tool for integrating Java with automation languages. As a part of these projects a general name analyzer for Java has been developed as a Jrag component. Additional ongoing projects using JastAdd involve translators for robot languages and support for extensible languages.

The JastAdd system is used in our department's undergraduate course on compiler construction. The students work in pairs and use JastAdd to implement a compiler for a small procedural language of their own design and producing SPARC assembly code as output. The course has covered both visitors and aspect-oriented programming using Jadd modules, but not Jrags or attribute grammars.

JastAdd is being bootstrapped in itself. This process has proceeded in several steps. Our starting point was the JavaCC/JJTree system which generates AST classes with untyped access methods and a simple default visitor. The first step was to implement the generation of AST classes with *typed* access methods to allow us to use visitors in a safer way. This step was itself bootstrapped by starting with hand coding the would-be generated AST classes for the abstract grammar formalism (a small amount of code), allowing us right away to use the typed access methods when analyzing abstract grammars. The next step was to use this platform (JJTree-generated visitors and our own generated AST classes with typed access methods) to implement the class weaving of Jadd modules. Once this was implemented we started to use Jadd modules for further implementation, adding the translator for Jrag modules (which generates a Jadd module), and improving the system in general. We are now continuing to improve the system and are also gradually refactoring it to use Jadd and Jrag modules instead of visitors.

The implementation of the JastAdd system is working successfully but we have many improvements planned such as generation of various convenience code, better error reporting, and extensions of the abstract grammar formalism.

There are several interesting ways to continue this research. One is to support modularization not only along phases, but also along the syntax. I.e., it would be interesting to develop the system so that it is possible to supply several abstract grammar modules that can be composed. Another interesting topic is to explore how dynamic aspect-modularization, for example using joinpoints in AspectJ, can be exploited in compiler construction. Yet another interesting direction is to investigate how emerging aspect-oriented techniques can be applied to achieve language-independent compiler aspects, e.g., name analysis and type analysis modules that can be parameterized and applied to many different abstract grammars. Work in this direction has been done by de Moor et al. for attribute grammars within a functional language framework [7]. We also plan to continue the development of reference attributed grammars and to applying them to new problem areas.

Acknowledgements

We are grateful to Anders Ive and to the anonymous reviewers for their constructive comments. Torbjörn Ekman and Anders Nilsson implemented the Java name analyzer. Many thanks also to the compiler construction students who provided valuable feedback on the system.

Bibliography

[1] ANTLR Translator Generator. Available at: http://www.ANTLR.org/.

[2] CUP. LALR Parser for Java. Available at: http://www.cs.princeton.edu/~appel/modern/java/cup/.

[3] JTB. Java Tree Builder. Available at: http://www.cs.purdue.edu/jtb/.

[4] The JavaCC Project, 2007. Available at: http://javacc.dev.java.net.

[5] E. Bjarnason, G. Hedin, and K. Nilsson. Interactive Language Development for Embedded Systems. *Nordic Journal of Computing*, 6(1):36–55, 1999.

[6] J. T. Boyland. *Descriptional Composition of Compiler Components*. PhD thesis, University of California, Berkeley, 1996.

[7] O. de Moor, S. Peyton-Jones, and E. Van Wyk. Aspect-oriented compilers. volume 1799 of *LNCS*, pages 121–133, 2000.

[8] B. B. Kristensen et al. Syntax-Directed Program Modularization. In *Integrated Interactive Computing Systems*. North-Holland Publishing Company, 1983.

[9] E. Gamma et al. *Design Patterns*. Addison-Wesley, 1995.

[10] G. Kiczales et al. Aspect-Oriented Programming. In *Proceedings of ECOOP 1997*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[11] E. M. Gagnon and L. J. Hendren. SableCC – an object-oriented compiler framework. In *Proceedings of TOOLS 26-USA'98, IEEE Computer Society*, 1998.

[12] W. Harrison and H. Ossher. Subject-Oriented Programming – a Critique of Pure Objects. In *Proceedings of 1993 Conference on Object-oriented Programming Systems, Languages and Applications*, 1993.

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

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

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

[16] M. Jourdan. An optimal-time recursive evaluator for attribute grammars. In *International Symposium on Programming*, volume 167 of *LNCS*, pages 167–178. Springer, 1984.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP 2001*, volume 2072 of *LNCS*, pages 327–355. Springer, 2001.

[18] J. L. Knudsen. Aspect-Oriented Programming in BETA using the Fragment System. In *Proceedings of the Ascpect-Oriented Programming WorkShop at (ECOOP'99)*, 1999.

[19] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In *Proceedings of LDTA'01*, 2001.

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

[21] O. L. Madsen. On defining semantics by means of extended attribute grammars. In *Semantics-Directed Compiler Generation, Proceedings of a Workshop*, volume 94 of *LNCS*, pages 259–299. Springer, 1980.

[22] O. L. Madsen and C. Nørgaard. An object-oriented metaprogramming system. In *Prodeedings of Hawaii Internat. Conf. on System Sciences*, volume 21, 1988.

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

[24] D. A. Watt and D. F. Brown. *Programming Language Processors in Java*. Prentice-Hall, 2000.

[25] B. Woolf. The null object pattern. In R. Martin et al., editor, *Pattern Languages of Program Design*, pages 5–18. Addison-Wesley, 1998.

PAPER III

CIRCULAR REFERENCE ATTRIBUTED GRAMMARS – THEIR EVALUATION AND APPLICATIONS

Abstract

This paper presents a combination of Reference Attributed Grammars (RAGs) and Circular Attribute Grammars (CAGs). While RAGs allow the direct and easy specification of non-locally dependent information, CAGs allow iterative fixed-point computations to be expressed directly using recursive (circular) equations. We demonstrate how the combined formalism, Circular Reference Attributed Grammars (CRAGs), can take advantage of both these strengths, making it possible to express solutions to many problems in an easy way. We exemplify with the specification and computation of the *nullable*, *first*, and *follow* sets used in parser construction, a problem which is highly recursive and normally programmed by hand using an iterative algorithm. We also present a general demand-driven evaluation algorithm for CRAGs and some optimizations of it. The approach has been implemented and experimental results include computations on a series of grammars including that of Java 1.2. We also revisit some of the classical examples of CAGs and show how their solutions are facilitated by CRAGs.

Published as: E. Magnusson, and G. Hedin, "Circular Reference Attributed Grammars – their Evaluation and Applications". SCP - Science of Computer Programming, 68(1):21-37. Elsevier. August 2007.

1 Introduction

Attribute grammars (AGs), as introduced by Knuth [19], allow computations on a syntax tree to be defined declaratively using attributes where each attribute is defined by a semantic function of other attributes in the tree. An attribute is either used to propagate information upwards in the tree (synthesized attribute) or downwards in the tree (inherited attribute). In the original form of AGs, the definition of an attribute may depend directly only on attributes of neighbor nodes in the tree. Furthermore, the dependencies between attributes may not be cyclic. The first of these restrictions is lifted by Reference Attributed Grammars (RAGs) [12] and similar formalisms, e.g., [3, 25]. In these formalisms, an attribute may be a reference to an arbitrarily distant node in the tree, and an attribute may be defined in a semantic function by directly accessing attributes of the reference (remote access). It has been shown earlier how RAGs support the easy specification and automatic implementation of many practical problems, for example, name- and type analysis of object-oriented languages [12], execution time prediction [24], program visualization [23], and design pattern checking [8].

The second of the restrictions mentioned above, circular definitions, is lifted by Circular Attribute Grammars (CAGs) such as those of Farrow [10] and Jones [15]. The traditional AG requirement of noncircularity is a sufficient but not necessary condition to guarantee that an AG is well defined in the sense that all semantic rules can be satisfied. It suffices that all attributes involved in cyclic dependencies have a fixed point that can be computed with a finite number of iterations. In CAGs, circular dependencies between attributes are allowed provided that such a fixed point is available for all possible trees. This is guaranteed if the values for each attribute on a cycle can be organized in a lattice of finite height and if all the semantic functions involved in computing these attributes are monotonic on the respective lattices. Several authors (e.g., [10], [15], [27]) have shown how the possibility of circular definitions of attributes allows simple AG specifications for some well-known problems from different areas. Examples include data-flow analysis, code optimizations, and properties of circuits in a hierarchical VLSI design system. Farrow [10] also demonstrates how alternative non-circular specifications in some cases can be constructed with additional huge complexity, including, e.g., the use of higher-order functions. The circular specifications, in contrast, are both easy to read and understand and easy for the AG author to write.

In this paper, we combine Circular Attribute Grammars (CAGs) and Reference Attributed Grammars (RAGs) into Circular Reference Attributed Grammars (CRAGs). We demonstrate how CRAGs can take advantage of both the combined formalisms, making it possible to express many new problems in a concise and straight-forward way. To exemplify, we show how to specify the *nullable*, *first*, and *follow* sets used in parser construction. These sets are traditionally defined using recursive equations and computed imperatively by iteration. We demonstrate in this paper how the recursive definitions can be expressed directly using

CRAGs. We also revisit some of the classical examples of CAGs, in particular, constant evaluation and live analysis, and show how their solutions are facilitated by CRAGs. We have developed a general recursive evaluation algorithm for CRAGs and implemented it in our tool JastAdd [13], which is an aspect-oriented compiler construction tool supporting RAGs. For evaluation, we present some experimental results of the CRAG evaluation of the *nullable*, *first*, and *follow* problems as compared to the corresponding handcoded iterative implementation.

There is some previous work on combining RAG-like formalisms with CAGs. Boyland has implemented a similar combination in his APS system [3]. Sasaki & Sassa present Circular Remote Attribute Grammars (also abbreviated CRAGs), which on the surface is similar to our CRAGs [27]. However, Sasaki & Sassa assume that the remote links are computed separately outside the attribute grammar.

The rest of this paper is structured as follows: Section 2 reviews existing evaluation algorithms for CAGs and RAGs. Section 3 introduces our demand-driven algorithm for CRAGs. In Section 4 we focus on some example applications and our experience of using CRAGs for their specifications. Section 5 summarizes the contributions and provides some directions for future work.

2 Existing Evaluation Algorithms

Dependencies between attribute instances in a syntax tree can be modelled as a directed graph. The vertices of the graph correspond to attribute instances and if the specification of an attribute a_1 uses another attribute a_2 there will be an edge from a_2 to a_1. If the dependency graph is acyclic for every possible derivable syntax tree for a certain grammar, the grammar is said to be noncircular. For noncircular grammars it is always possible to topologically order the dependency graphs and evaluate the attributes by applying the semantic functions in that order.

Traditional AGs are required to be noncircular, but, as has been shown by, e.g., Farrow [10] and Jones [15], grammars with circular dependencies under certain constraints can be considered well defined in the sense that it is possible to satisfy all semantic rules for all possible syntax trees. One way to formulate the constraints is to require that the domain of all attributes involved in cyclic chains can be arranged in a lattice of finite height and that all semantic functions for these attributes are monotonic.[1] The evaluation of circularly defined attributes can be regarded as a special case of solving the equation $X = f(X)$ for the value of X. By giving X the bottom of the lattice as start value the iterative process $X_{i+1} = f(X_i)$ will converge to a least fixed point for which all involved semantic rules are satisfied.

[1] More precisely it is sufficient that the domain of the attributes forms a complete partial order and that the semantic functions satisfy the ascending chain condition.

The values of the attributes involved in a cycle can be computed by the iterative algorithm shown in Fig. 1. The arguments of the semantic functions f_i are to be the values from the previous iteration of all attributes on which x_i depends.

```
initialize all attributes x_i involved in the cycle to a bottom value;
do {
    foreach attribute x_i in the cycle
        x_i = f_i(...);
} while (some computation changes the value of an attribute);
```

Figure 1: Iterative algorithm for computing the least fixed point for attributes on a cycle.

2.1 Detection of circularity

The problems of deciding whether a traditional AG is circular and of identifying the attributes taking part in cycles have been addressed by several researchers. In [19] Knuth developed a polynomial algorithm for circularity testing. The algorithm is conservative, i.e., circularity is always detected but some noncircular AGs may be reported to be circular. Later [20] Knuth constructed an exact algorithm which is exponential in time and space complexity. Rodeh & Sagiv [26] extended these algorithms to deal with the problem of finding circular attributes. They developed a polynomial approximation algorithm, i.e., all circular attributes are discovered but some noncircular attributes may be reported to be circular. They also constructed an exact algorithm with exponential time complexity and showed that finding the circular attributes is a harder problem than testing for circularity. The problem of testing reference attributed grammars for circularity can be addressed as in [27] by introducing all possible remote edges with lots of potential cycles in the dependency graph as a result. Most abstract syntax trees will, however, not contain any cycles and the iterations performed by the generated evaluator are unnecessary. In [3] Boyland has defined an extension to traditional AGs called remote attribute grammars, which supports reference attributes like in RAGs and also additional features like collection attributes that can be written from remote locations. In [7] he shows that testing remote attribute grammars for circularity is undecidable and examines techniques for approximation.

2.2 Evaluation of circular attribute grammars

Jones [15] proposes a dynamic evaluation algorithm derived from the underlying attribute dependency graph. Optimal dynamic evaluation for circular AGs is obtained by analyzing the dependency graph dynamically to identify its strongly

connected components. A strongly connected component is a maximal set of vertices in which there is a path from any one vertex in the set to any other vertex in the set. All attribute instances belonging to the same strongly connected component are thus dependent of each other. Each strongly connected component is contracted into a single node to obtain a new graph `C(G)`, which is acyclic and can be ordered topologically and evaluation can follow this order. A vertex in `C(G)` corresponding to more than one vertex in the original graph represents a set of attribute instances that are all dependent of each other and they will be evaluated in a single fixed-point evaluation. The graphs must be constructed initially. When the attribute grammar is acyclic, Jones' algorithm reduces to a standard optimal algorithm for noncircular evaluation. His scheme is not immediately applicable to RAGs since the reference attributes introduce dependencies that are not known until they have been evaluated.

Farrow [10] introduced a static evaluation technique based on the one by Katayama [18], but modified to compute the fixed point for attributes which potentially have circular dependencies. His scheme is also limited to traditional AGs without remote references since it depends on deriving the attribute dependencies statically from the productions of the grammar. Sasaki & Sassa [27] have elaborated on the technique of Farrow in the presence of remote references. However, these references are not considered to be a part of the attribute grammar and must be evaluated separately in an initial phase. They also make the additional assumption that cycles do not appear without remote references, a constraint that facilitates the check for convergence.

The static evaluation technique used by Farrow and Sasaki & Sassa is realized with a group of mutually recursive functions along the AST. Inefficiency arises when iterative evaluation of a group of attribute instances includes other iterative evaluations further down the tree. Fig. 2 illustrates this: Attribute instances belonging to a strongly connected component with more than one vertex are indexed, e.g., a_1, a_2, a_3 and a_4, and the corresponding component will be called A. Consider case (I). An iterative evaluation of the four a_i attributes will in each iteration call a function evaluating the b_i attributes belonging to another cyclic component B. A new iterative process will thus be started bringing B to a fixed point in each iteration of A. Case (II) gives rise to the same kind of inefficiency.

Sasaki & Sassa have shown how to overcome this shortcoming and avoid inner loops by using a global variable to keep track of whether the computation is already within an iterative phase. Iterations will in their case, as a consequence, take place over a larger number of attribute instances belonging to more than one strongly connected component of the dependency graph. For case (I), iterations will span over components A and B, and in case (II) components A, B, and C will be part of the same iterative process.

The static techniques of Farrow as well as that of Sasaki & Sassa have another shortcoming in that iterative evaluation will include a possibly large number of noncircular attribute instances below the AST node associated with the circularly

defined attributes that started the iterative process. Case (III) in Fig. 2 is an example. The noncircular attributes b, c and d will then be evaluated during each iteration of the evaluation of component A.

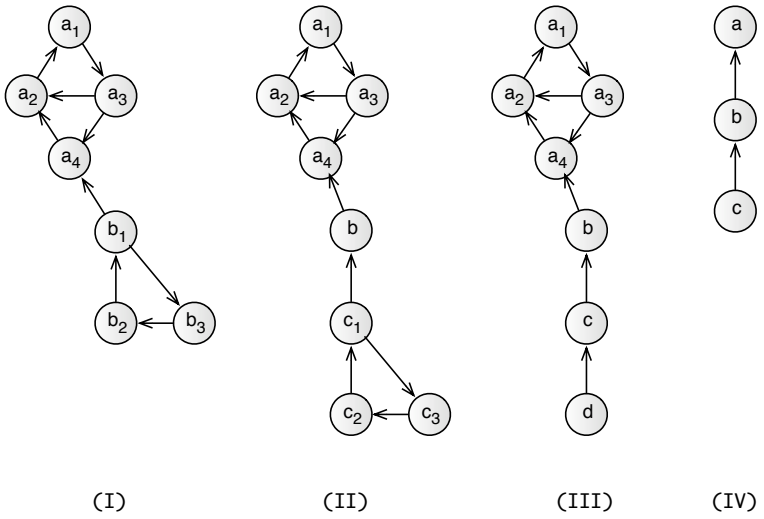

Figure 2: Different cases of dependencies involving strongly connected components.

Non-monotonic intercomponent dependencies

If two strongly connected components are evaluated in topological order, the dependency between the components does not need to be monotonic. The inner component will then have received its final value before it is used by the outer component, and the monotonicity of the intercomponent dependency is therefore not important. This was pointed out to us by Boyland, who also reports practical uses of such nonmonotonic dependencies in, e.g., using inferred types in polymorphic type checking [6].

2.3 Demand-driven evaluation of AGs

We will base the evaluation of CRAGs on a general demand-driven evaluator for non-circular AGs where each attribute is implemented by a method that recursively calls the methods implementing other attributes. By caching evaluated attribute values in the syntax tree, the evaluator is optimal in that it evaluates each attribute at most once. (Our experimental system allows the user to choose which

attributes are to be cached. In the rest of this paper we will, however, assume that all attributes are cached in order to achieve optimality.) Circular dependencies can be checked at evaluation time by keeping track of which attributes are being evaluated. In principle, this evaluator is the same as the ones used for traditional AGs by Madsen [21], Jalili [14], and Jourdan [16], although we use an object-oriented implementation [11, 13]. The evaluator is dynamic in that dependencies are not analyzed statically. In fact, the dependencies between attributes need not be analyzed at evaluation time either since the call structure of the recursive evaluation automatically results in an evaluation in topological order. Thus, in contrast to other dynamic evaluation algorithms, no explicit dependency graph is built. The demand-driven dynamic algorithm is sufficiently fast for practical use, also for large examples. Current experiments with generated Java compilers indicate that this kind of evaluation method easily performs within a factor of 3 from handwritten compilers [9]. Our evaluator is implemented in Java which provides a straight-forward implementation of the algorithm. Fig. 3 shows a fragment of an AG based on abstract syntax and the corresponding evaluator code in Java.

AG	Evaluator code

	abstract class Node {
	  Node ancestor;
	}

Exp {
  syn int val;
}
	abstract class Exp **extends** Node {
	  int val_value;
	  boolean val_computed = false;
	  boolean val_visited = false;
	  abstract int val();
	}

AddExp: Exp ::= Exp_1 Exp_2 {
  val = Exp_1.val + Exp_2.val;
}
	class AddExp **extends** Exp {
	  Exp exp1, exp2;
	  int val() {
	  if (val_computed) **return** val_value;
	  if (!val_visited) {
	  val_visited = true;
	  val_value = exp1.val() + exp2.val();
	  val_computed = true;
	  val_visited = false;
	  return val_value;
	  }
	  else throw new RuntimeException
	  (”Circular definition of attribute”);
	  }
	}

Figure 3: Demand-driven evaluator for noncircular AG.

The abstract syntax is translated to classes and fields modelling an abstract

syntax tree (AST). A general class `Node` models the general aspects of all AST nodes. For instance, each AST node has an ancestor node. Each nonterminal, like `Exp`, is translated to an abstract class, and each of its productions, like `AddExp`, is translated to a concrete subclass. A right-hand side is translated to fields in the production class (e.g., `Exp exp1, exp2;`).

Each synthesized attribute declaration (e.g., `syn int val`) is translated to an abstract method specification (e.g., `abstract int val();`), a field for storing the cached value (e.g., `val_value`), and two additional boolean fields for keeping track of if the attribute is already computed (`val_computed`) and if it is under computation (`val_visited`). Each equation that defines a synthesized attribute is translated to a corresponding method implementation (e.g., `int val() {...}`). If the value is already computed, the method simply returns the cached value. If not, it computes the value, which involves calling methods corresponding to other attributes (e.g., `val_value := exp1.val() + exp2.val();`). The `val_visited` field is used in order to check for circular dependencies, thereby avoiding endless recursion, and an exception is raised if a circularity is found.

Inherited attributes are implemented in a similar, although slightly more involved, manner, making use of the ancestor field to call methods of the ancestor node. See [13] for details.

2.4 Demand-driven evaluation of RAGs

RAGs can be evaluated using the same demand-driven algorithm as for AGs with the extension of allowing attributes to be references to other nodes in the AST [13]. A typical use of reference attributes is in name analysis, where applied occurrences of identifiers are linked to declared occurrences. Fig. 4 shows fragments of a typical RAG with such links. For example, the `IdExp` production contains a reference attribute `decl` which is a reference to the appropriate `Decl` node in the AST. The implementation of the evaluator is a straight-forward extension of the demand-driven AG evaluator. An access to a reference attribute is translated to a call to the corresponding method computing the reference value. For example, the `decl()` method in `IdExp` computes the `decl` reference value. This is done by first computing the value of the `env` attribute (also a reference attribute) and then calling the `lookup` method of the `env` object.

The example also illustrates a number of additional features of RAGs: A production may occur directly in the right-hand side of another production. E.g., `Decl` is used in the right-hand side of `Block`. General nonterminals that do not appear on any right-hand side are allowed (e.g., `Any`). These can be used to capture attributes and equations applying to many other classes, e.g., the `env` attribute. The right-hand sides may contain lists (as in `Block`) or `String` tokens (like `<ID>`). Classes in the RAG may contain ordinary methods in addition

RAG	Evaluator code

Any {
  inh Block env;
}
	class Any **extends** Node {
	  Block env() {
	  ...
	  }
	}

Block: Any ::= Decl* Stmt* {
  Decl lookup(String name) {
  ...
  }
}
	class Block **extends** Any {
	  List decls;
	  List stmts;
	  Decl lookup(String name) {
	  ...
	  }
	}

Decl: Any::= Type <ID> {
}
	class Decl **extends** Any {
	  Type type;
	  String ID;
	}

Exp: Any {
  syn Type tp;
}
	class Exp **extends** Any {
	  abstract Type tp();
	}

IdExp: Exp ::= <ID> {
  syn Decl decl =
  env.lookup(<ID>);
  syn Type tp =
  decl != null ? decl.type : null;
}
	class IdExp **extends** Exp {
	  String ID;
	  ...
	  Decl decl() {
	  ...
	  decl_value = env().lookup(ID);
	  ...
	  }
	  Type tp() {
	  ...
	  tp_value =
	  decl() != null ? decl().type : null;
	  ...
	  }
	}

Figure 4: Example of RAG and corresponding evaluator.

to attributes (like `lookup` in `Block`). These methods must be side-effect free, however.

3 An evaluator for CRAGs

We now turn to CRAGs and their evaluation. The CRAG fragment in Fig. 5 declares a synthesized set-valued attribute `s`. The attribute is explicitly declared as `circular` and the bracketed expression encloses the bottom value (an empty set in this case).

CRAG	Evaluator code

A {
  syn Set s **circular** [**new** Set()];
}
	abstract class A **extends** Node {
	  Set s_value = **new** Set();
	  boolean s_computed = false;
	  boolean s_visited = false;
	  abstract Set s();
	}

B: A ::= ... {
  s = f(...)
}
	class B **extends** A {
	  ...
	  Set s() { ... }
	}

Figure 5: Example of CRAG fragment and corresponding classes.

3.1 Basic algorithm

We will now extend the demand-driven evaluator from Sections 2.3 and 2.4 to handle CRAGs. Fig. 6 shows a basic evaluation algorithm for the circular attribute `s`. This algorithm is a straight-forward implementation of the iterative process shown in Fig. 1.

The algorithm makes use of two global variables: `IN_CIRCLE` keeps track of whether we are already inside a cyclic evaluation phase. When this is the case, `CHANGE` is used to check whether any changes of iterative values of the attributes on the cycle have taken place during an iteration. The right-hand sides of the two assignment statements for `new_s_value` are the expressions corresponding to the semantic function for the attribute `s`. It thus involves calls for evaluation of attributes on which `s` is dependent, some of which will be in the same cycle as `s`.

```
class B extends A {
    ...
    Set s() {
        if (s_computed) return s_value;
        if ( ! IN_CIRCLE) {
            IN_CIRCLE = true;
            s_visited = true;
            do {
                CHANGE = false;
                Set new_s_value = f(...);
                if ( ! new_s_value.equals(s_value))
                    CHANGE = true;
                s_value = new_s_value;
            } while (CHANGE);
            s_visited = false;
            s_computed = true;
            IN_CIRCLE = false;
            return s_value;
        }
        else if ( ! s_visited ) {
            s_visited = true;
            Set new_s_value = f(...);
            if ( ! new_s_value.equals(s_value))
                CHANGE = true;
            s_value = new_s_value;
            s_visited = false;
            return s_value;
        }
        else
            return s_value;
    }
}
```

Figure 6: Evaluation code for the equation $s = f(...)$ where s is a circular attribute.

3.2 Comparison of algorithms

In this and the following subsections we will compare our algorithm to existing algorithms and also present some improvements of the basic algorithm shown in Fig. 6 in order to avoid certain inefficiencies.

To facilitate the description we will use the following terminology: An attribute is *definitely noncircular* if no instance of the attribute can be part of a cycle in the dependency graph for any derivable AST. An attribute is *potentially circular* if some instance can be part of a cycle for some AST. An instance of a potentially circular attribute in a certain AST is *actually circular* if it is on a cycle and otherwise *actually noncircular*.

All potentially circular attributes are required to be declared `circular`. (In Section 3.4 we will discuss how to detect and handle failures of this requirement.) Thus, we have a similar situation as in Farrow's static evaluator where potentially circular attributes are detected by analyzing the productions of the grammar. However, some of the shortcomings of the static technique mentioned in Section 2.2 are avoided by our basic algorithm and others can be avoided by small modifications of our demand driven evaluator given the possibility to cache attribute values.

We will use the different cases of Fig. 2 in the discussion below.

Nested iterative evaluations are avoided

In Farrow's static method and in the basic method of Sasaki & Sassa, an iterative evaluation may recursively include another iterative evaluation. The number of iterations in the innermost loop becomes an exponential factor of its nesting level. Sasaki & Sassa improve their evaluator to avoid such nested behavior by introducing a global variable. In our evaluator the global variable `IN_CIRCLE` achieves the same improvement. However, as a consequence, iterations might span over more than one strongly connected component of the dependency graph. This is suboptimal behavior as compared to the dynamic algorithm of Jones, where each component is evaluated individually. In Section 3.3 we will show how this inefficiency can be avoided in some cases.

Iterative evaluation of definitely noncircular attributes is avoided

Recall case (III) of Fig. 2, and assume that b is definitely noncircular. Suppose that one of the attribute instances of component A is demanded. An iterative process is then started during which b will be demanded. Since b is cached it will only be evaluated the first time it is demanded. When a later iteration in component A demands b again, its computed value will be returned. This differs from the static evaluation techniques of Farrow and Sasaki & Sassa, where definitely noncircular attributes might be evaluated during each iteration.

3.3 Improving the algorithm

We can avoid some additional inefficiencies by slight modifications to our demand-driven evaluator.

Avoiding recomputation of potentially circular attributes

The basic algorithm in Fig. 6 computes the value of an attribute `s` and caches the intermediate values of the circular attributes involved in the cycle. When the iterative evaluation has converged, the attribute `s` has reached its fixed point and is registered as computed by setting the field `s_computed`. However, at this point, all other attributes on the cycle have reached their fixed point as well, but are not registered as computed. For efficiency reasons it is desirable to register these attributes as computed in order to avoid their recomputation in case they will be demanded again. By introducing another global variable `READY`, that is set to true when the fixed point is reached, it is possible to perform one extra iteration during which all involved attribute instances register themselves as computed.

Evaluating strongly connected components in topological order

Consider case (II) of Fig. 2 and suppose that b is definitely noncircular. When an attribute of component A is demanded, an iterative process is started and eventually b will be demanded. b will in turn demand c_1. A new strongly connected component is thereby entered, but a new iterative process would not be started by the basic algorithm shown in Fig. 6 since `IN_CIRCLE` is already `true`. The resulting iterative process would thus involve all attributes of components A, B, and C just as in the static techniques mentioned in Section 2.2. It would be more efficient to suspend the iterative process of A temporarily and start a new iterative process for component C, and thereby avoid unnecessary evaluations in A while the attributes in C are being computed. This scheme can be realized by slightly modifying the algorithm for definitely noncircular attributes (i.e. the algorithm in Fig. 3). An outline of the modified algorithm is given in Fig. 7. When the attribute b is demanded, the status of the iterative process is now stacked (`CHANGE` flag), b calls its semantic function and on return, the interrupted cyclic evaluation of component A is resumed. When b demands the attribute c_1 a new cycle is entered, so the component C will be brought to a fixed point before b gets its value. When b is computed, the suspended iterations of A are resumed. Since all cyclic attributes are cached after they have been brought to a fixed point, the attributes in cycle C will only be computed once.

Avoiding iterative evaluation of actually noncircular attributes

For many ASTs there might be many actually noncircular instances of potentially circular attributes. Consider case (IV) in Fig. 2 and suppose a is demanded. If

```
boolean interrupted_Circle = false;
if (attribute_computed)
    return attribute_value;
if ( ! attribute_visited ) {
    attribute_visited = true;
    if (IN_CIRCLE) {
        push value of CHANGE on stack;
        IN_CIRCLE = false;
        interrupted_Circle = true;
    }
    attribute_value = f(..);
    attribute_computed = true;
    if (interrupted_Circle) {
        CHANGE = pop from stack;
        IN_CIRCLE = true;
    }
    attribute_visited = false;
    return attribute _value;
}
else throw new RuntimeException("Circular def...");
```

Figure 7: Pseudo-code for improved evaluation of a definitely noncircular attribute.

a is potentially circular, an iterative process is started in which b and c will be demanded. Again, a small modification of the algorithm makes it possible to detect that no cycle is ever encountered and interrupt the iterative process. Basically, a global variable is used to keep track of if we have encountered an already visited attribute during an ongoing iterative evaluation process.

Sasaki & Sassa [27] also have a refined mostly static version of their originally completely static technique. The basic idea is here to have several versions of attribute evaluation sequences, one for each possible pattern of remote dependency edges. The actual pattern for each subtree in the AST is then computed at runtime and the evaluator selects the proper version. If there are no actually circular attributes in a subtree, iterations are avoided for the production at its root, provided cycles are always caused by remote references. It is not clear if their algorithm can be generalized to deal with cyclic behavior that is not caused by remote links. The refinement deals only with potentially circular attributes that are not actually circular. Their evaluator will still make unnecessary iterations for definitely noncircular attributes in a subtree below AST nodes corresponding to productions with actually circular attributes.

3.4 Robust improved algorithm

So far, we have assumed that the AG author has declared all potentially circular attributes as `circular`. As will become evident from examples in Section 4, it is often apparent to the AG author which attributes are potentially circular. However, if the AG author has forgotten to declare an attribute as `circular`, and it is in fact actually circular, the algorithms in figures 6 and 7 may yield erroneous results. Consider Fig. 8 as an example. There are five attribute instances of which four (a, b, c, and d) have a circular dependency. Given the equations to the right in the figure, it is obvious that the set `{id}` should be the final value of all attributes after a fixed-point iteration. Suppose that the AG author has forgotten to declare attribute c as circular and suppose that attribute a is demanded. An iterative process is started, b is demanded and then c is demanded. Since c is not declared circular its evaluation code will be that of Fig. 7 and thus the iterative phase will be temporarily suspended and d will be demanded. Since d is a circularly declared attribute, a new iterative process is started. When a is demanded it is already visited, so it will return its current value (the bottom value). The iterative process started by d will thus only involve attributes d and a and their values will never change from the bottom value, i.e, the empty set. Consequently the value of c will also be the empty set. The interrupted iterative process started by the evaluation of a is resumed when c has been evaluated. Since c is cached, the iterations will only span over attributes a and b and the fixed point is reached when their respective values are `{id}`. Obviously all semantic rules are not satisfied.

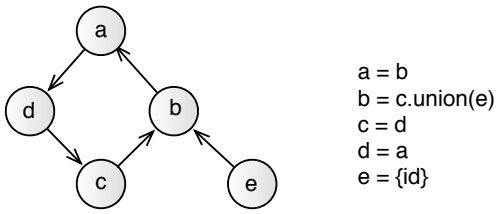

Figure 8: Equations for some attributes creating cyclic dependency.

In order to make the algorithm robust to such grammar errors, the algorithm can be modified as follows. Using the information about which attributes are declared circular, it is possible to keep track of which nodes in the dependency graph might belong to the same strongly connected component. If a visited node belonging to another component is encountered, then an error has ocurred. In the case described above the evaluator would consider attributes a and b to belong to one component and attribute d to another. When the evaluation of d demands a, a visited node belonging to a different component is encountered. The scheme of

keeping track of components can be realized by adding vertices of the dependency graph to a set during evaluation, as long as only potentially circular attributes are encountered. This set will be stacked together with the `CHANGE` flag when an iteration is temporarily suspended as in Fig. 7. When a visited node is encountered it can then be checked if it belongs to the set of the component actually being brought to a fixed point. Thus, in case of a missing circular declaration, the algorithm will detect the error, identify the attributes involved, and raise an exception.

3.5 Comparison to related work

Our evaluation algorithm uses a pure dynamic demand-driven technique where no initial dependency analysis is performed. In general, the complete dependency graph for a RAG or a CRAG is not known until *after* evaluation, since the dependencies introduced by reference attributes will depend on the reference values.

Boyland takes a similar approach in his APS system where he has implemented support for evaluation of circular attributes for remote attribute grammars. His evaluation method is based on demand-driven evaluation and performs topological evaluation of strongly-connected components, allowing non-monotonic dependencies between components. However, he provides no explicit evaluation algorithms [3].

For ordinary attribute grammars, the dependency graph can be computed from the grammar and constructed before evaluation. The static attribute evaluation algorithms available for ordinary attribute grammars, like OAGs [17], rely on this property in order to compute approximations of the dependency graph before evaluation. The same holds for the static evaluation algorithms for circular attribute grammars, like Farrow's algorithm [10].

The development of static evaluators for subcategories of RAGs and CRAGs is a problem that we have not pursued, but there is some other work in this direction. In [4], [5], [7], Boyland addresses the problem of analyzing (noncircular) non-local dependencies statically. He develops a technique to schedule evaluation statically and shows how it may be implemented incrementally. The technique has been applied to variants of name and type analysis including the static-semantics of a simple object-oriented language.

Sasaki & Sassa [27] allow circular dependencies as well as remote links between nodes in the AST, but links between nodes are not considered a part of the AG and must be provided by a separate initial phase that they have not elaborated further on in their paper. In contrast, our demand-driven evaluation technique allows reference attributes as well as ordinary attributes to be evaluated in the same manner. An additional constraint in the scheme of Sasaki & Sassa is that cycles are assumed to arise only from remote references.

As was discussed in Section 3.2, the static evaluation algorithms of Farrow and of Sasaki & Sassa have suboptimal behavior for strongly connected components of circular attributes, while the dynamic algorithm of Jones [15] is optimal. Jones

algorithm computes the strongly connected components for a given AST before evaluation, based on the actual attribute dependencies. For CRAGs, such pre-evaluation computations are not possible in general, since the reference attributes give rise to attribute dependencies that are not known until after the evaluation of those reference attributes. For this reason, the strongly connected components in CRAGs cannot, in general, be computed before evaluation. Our evaluation algorithm detects strongly connected components during evaluation, and does not always have sufficient information to distinguish between two components. In this case, the components will be evaluated together rather than in topological order, thereby yielding suboptimal evaluation. However, in Section 3.3 we showed how to avoid many of these cases by using cached attribute values. The remaining suboptimal case is the one where there are two adjacent strongly connected components. I.e., where an attribute in one component depends directly on the attribute in another component, like in case (I) of Fig. 2. To handle also these cases, there would be the need for either more information given by the grammar author, or for some kind of approximative pre-evaluation analysis. The development of such analyses remains an open issue. For non-circular RAGs, dynamic demand-driven evaluation using caching is optimal (each attribute is evaluated at most once).

For CRAGs, we rely on the author to declare a potentially circular attribute as `circular`, which provides the same information as the analysis of the grammar performed initially in the static methods of Farrow and Sasaki & Sassa. In both cases the potentially circular attributes are identified and become known to the evaluator. A less experienced author might forget to declare some attributes that are potentially part of cyclic dependencies as `circular`. Our evaluator will then report an error on inputs where cycles do appear and it will produce a correct result on cycle-free input.

We find it reasonable to demand of the grammar author to explicitly declare which attributes are potentially circular. The grammar author needs to be aware of such attributes since they should be given a start value (bottom of the lattice), and their semantic functions must be monotonic. In principle, it would be possible to instead regard *all* attributes as potentially circular, and use default start values. However, this would imply that attributes that were mistakenly defined in a circular manner might lead to nonterminating evaluation, e.g. if the functions were non-monotonic. In our system, such mistaken circularities would be flagged as errors at evaluation time. To regard attributes as potentially circular when they are in fact definitely noncircular, would also lead to performance degradation. We would not be able to perform the optimizations described above that make use of knowing which attributes are definitely noncircular.

Our evaluator presently does not check whether circularly defined attributes take their values from a lattice of finite height or if their defining semantic functions are monotonic. Thus there is no guarantee that iterations will converge. Our approach is in this respect similar to that of, e.g., Farrow [10] and means that we rely on the AG author to ensure that the semantic functions involved are properly

constrained.

4 Application Examples

In this section we will discuss three examples which are naturally expressed using recursion and circular dependencies. Two of them are classical and are discussed in earlier papers dealing with circular attribute grammars. In these cases we will focus on a comparison between the solutions proposed earlier and solutions made possible when reference attributes are available. However, we start with an example that computes *nullable*, *first*, and *follow* in the context of parser construction. This is a problem that, to our knowledge, has not been solved using an attribute grammar approach before. This application is typical for a large class of problems within compiler construction that deal with computing various properties of grammars. Other similar problems are the computation of static dependency graphs in the context of attribute grammars, computation of visit sequences for ordered attribute grammars, etc. All these problems are expressed as highly recursive equations and are typically solved by iterative fixed-point computations.

4.1 Computation of nullable, first and follow

Given a context-free grammar (CFG), a recursive-descent or predictive parser can be generated if the first terminal symbol of each subexpression provides enough information to select production. This can be more precisely formulated by introducing the notion of a nonterminal being *nullable* and by defining the sets *first* and *follow*, informally defined as:

- A nonterminal *X* is *nullable* if the empty string can be derived from *X*.
- *first(X)* is the set of terminals that can begin strings derived from *X*.
- *follow(X)* is the set of terminals that can immediately follow *X*.

Fig. 9 shows an example context-free grammar and its values for *nullable*, *first*, and *follow* (grammar 3.12 in Appel [2]).

Nonterminals and their productions	*nullable*	*first*	*follow*
X → Y | a	true	{a,c}	{a,c,d}
Y → c | ϵ	true	{c}	{a,c,d}
Z → XYZ | d	false	{a,c,d}	$\emptyset$

Figure 9: Example CFG and its values for $nullable$, $first$, and $follow$.

Computation of nullable

The following equations hold for *nullable*:

(i) Let X be a nonterminal with the productions $X \rightarrow \gamma_1, X \rightarrow \gamma_2, ... X \rightarrow \gamma_n$
X is nullable if any of its production right-hand sides is nullable:
$nullable(X) == nullable(\gamma_1) \,||\, nullable(\gamma_2), ... \,||\, nullable(\gamma_n)$

(ii) Let ϵ be an empty sequence of terminal and nonterminal symbols.
The empty sequence is nullable:
$nullable(\epsilon) == true$

(iii) Let $\gamma = s\delta$ be a nonempty sequence of terminal and nonterminal symbols
where s is the first symbol and δ is the remaining (possibly empty) sequence.
γ is nullable if both s and δ are nullable:
$nullable(\gamma) == nullable(s)\ \&\&\ nullable(\delta)$

(iv) A terminal symbol t is not nullable:
$nullable(t) == false$

These equations are circular which is evident from (i) since X might be identical to, or derivable from, one of the nonterminal symbols on the right-hand side of one of the productions. The domain of *nullable* is a boolean lattice with the bottom value *false*. The functions corresponding to the right-hand side of the equations are monotonic with respect to this lattice. Therefore an iterative process initializing *nullable* for each symbol of the grammar to *false* will compute the least fixed point of the equations.

The above equations can, with trivial adaptions to syntax form, be formulated directly in a CRAG as demonstrated in Fig. 10. The CRAG equations corresponding to (i) - (iv) above are marked in the CRAG specification. In the CRAG, we differ between declared and applied occurrences of nonterminal symbols (`NDecl` and `NUse`). Each `NUse` is bound to the appropriate `NDecl` by means of a reference attribute `decl` which is specified in a similar way as was sketched in Section 2.4. Their values for `nullable` are equal as indicated by equation (v).

Computation of first

The following equations hold for the *first* set for symbols and symbol sequences:

(i) Let X be a nonterminal with the productions $X \rightarrow \gamma_1, X \rightarrow \gamma_2, ... X \rightarrow \gamma_n$
$first(X) == first(\gamma_1) \cup first(\gamma_2) ... \cup first(\gamma_n)$

(ii) Let ϵ be an empty sequence of terminal and nonterminal symbols.
$first(\epsilon) == \emptyset$

(iii) Let $s\delta$ be a nonempty sequence of terminal and nonterminal symbols
where s is the first symbol and δ is the remaining (possibly empty) sequence.
$first(s\delta) ==$ **if** $(nullable(s))$ **then** $first(s) \cup first(\delta)$ **else** $first(s)$

(iv) Let t be a terminal symbol.
$first(t) == \{t\}$

The equation system is circular which is evident from (i) since X might be identical to, or derivable from, one of the nonterminal symbols on the right-hand side

```
CFG ::= Rule * { }
Rule ::= NDecl ProdList {
    NDecl.nullable = ProdList.nullable;                          (i)
}
NDecl ::= <ID> {
    inh boolean nullable circular [false];
}
ProdList, Prod, SymbolList, Symbol {
   syn boolean nullable circular [false];
}
EmptyProdList: ProdList ::= {
   nullable = false;                                             (i)
}
NonEmptyProdList: ProdList ::= Prod ProdList {
   nullable = Prod.nullable || ProdList.nullable;                (i)
}
Prod ::= SymbolList {
   nullable = SymbolList.nullable;                               (iii)
}
EmptySymbolList: SymbolList ::= {
   nullable = true;                                              (ii)
}
NonEmptySymbolList: SymbolList ::= Symbol SymbolList {
   nullable = Symbol.nullable && SymbolList.nullable;            (iii)
}
Terminal: Symbol ::= <TERMINAL> {
   nullable = false;                                             (iv)
}
NUse: Symbol ::= <ID> {
   syn NDecl decl = ...;
   nullable = decl.nullable;                                     (v)
}
```

Figure 10: A CRAG that computes *nullable*.

of one of the productions. We can also note that the equations for *first* relies on the values of *nullable*. The domain of *first* is the lattice of finite subsets of the set of all terminals of the grammar with the empty set as the bottom value. The expressions of the right-hand side of the equations are monotonic with respect to this lattice. Figure 11 shows the corresponding CRAG including (i) - (iv) from the equations above. As in the case of *nullable*, the *first* computation relies on the `decl` reference attribute in `NUse` to equate the `first` values of an `NUse` and its corresponding `NDecl` (v).

```
CFG ::= Rule * { }
Rule ::= NDecl ProdList {
    NDecl.first = ProdList.first;                                    (i)
}
NDecl ::= <ID> {
    inh Set first circular [∅];
}
ProdList, Prod, SymbolList, Symbol {
   syn Set first circular [∅];
}
EmptyProdList: ProdList ::= {
   first = ∅;                                                        (i)
}
NonEmptyProdList: ProdList ::= Prod ProdList {
   first = Prod.first ∪ ProdList.first;                              (i)
}
Prod ::= SymbolList {
   first = SymbolList.first;                                         (iii)
}
EmptySymbolList: SymbolList ::= {
   first = ∅;                                                        (ii)
}
NonEmptySymbolList: SymbolList ::= Symbol SymbolList {
    first = Symbol.nullable                                          (iii)
              ? Symbol.first ∪ SymbolList.first
              : Symbol.first;
}
Terminal: Symbol ::= <TERMINAL> {
   first = { <TERMINAL> };                                           (iv)
}
NUse: Symbol ::= <ID> {
   first = decl.first;                                               (v)
}
```

Figure 11: A CRAG that computes *first*.

Computation of follow

The definition and CRAG for *follow* is similar in style to *nullable* and *first*, but makes additional use of reference attributes: To compute *follow* for a nonterminal X we need to locate all the applied occurrences of X and look at the subsequent symbols. To this end, reference attributes are used for linking an `NDecl` to all its `NUses`. The additions of such cross-referencing attributes are straight-forward using RAGs, for example by defining a set of `NUse` references at each `NDecl`. These set-valued attributes can easily be defined using parameterized attributes which are analogous to virtual functions, and which are available in RAGs [12]. In essence, such a function simply traverses a suitable portion of the AST to collect the appropriate information. An example of such a computation is shown in our paper on using RAGs for visualization computations [22]. The collection attributes of Boyland [3] would provide a more elegant way of defining such cross-reference information. With these cross-reference attributes in place, the specification of $follow$ becomes as straight-forward as for *nullable* and *first*, and is also circular.

During evaluation, the computation of *nullable*, *first*, and *follow*, forms three strongly connected components where the *first* component depends on the *nullable* component, and the *follow* component depends on both the *nullable* and *first* components.

Experimental results

We have implemented the robust improved CRAG evaluation algorithm in our compiler construction tool JastAdd. In order to test performance, we developed a CRAG for computing *nullable*, *first*, and *follow* for context-free grammars. We have compared the generated CRAG evaluator with a typical hand-coded iterative implementation. We have tried to make the basis for comparison as fair as possible: Both implementations use the same implementation language (Java), the same underlying AST classes, and the same data structure classes (for sets etc.). There has been no effort put into optimizing any data structures or operations. All is implemented in a straight-forward manner using classes, objects, and methods.

The results are shown in Fig. 12. The grammars Appel 1 and Appel 2 are small example grammars from [2]. Appel 1 is a toy language (the same as in Fig. 9.) with 3 nonterminals (#N) and 6 productions (#P) and Appel 2 is a grammar for simple arithmetic expressions. Tiny is a grammar for a small block-structured language. The grammar for Java 1.2 is the largest and has been taken from the examples distributed with JavaCC [1]. It has about 160 nonterminals when written in our CFG language. The times given are average times for 100 executions on a Sun Ultra 80 using the HotSpot JVM. The results indicate that the evaluator of CRAG performs as well as the handwritten iterative evaluation code. For a large grammar like Java the declarative approach even seems to be superior. One explanation could be that in an imperative style fixed-point iteration, the order in which the productions are processed is very important. (See, e.g. [2] chapter 17.4.) The

CRAG evaluator, on the other hand, traverses the dependency graph depth first, i.e., in topological order, and the iterations will thus usually converge faster.

We can also see that the maximum number of iterations for a single attribute value to converge (#I-A) seems to be almost constant, regardless of grammar size, whereas the total number of iterations (#I-T) naturally depends on the number of attributes, and thereby on the size of the grammar.

			CRAG			**Handwritten**
Language	$\#N$	$\#P$	$\#I\text{-}T$	$\#I\text{-}A$	time (ms)	time (ms)
Appel 1	3	6	8	4	8	7
Appel 2	6	12	18	4	13	9
Tiny	18	30	35	4	22	15
Java 1.2	157	321	263	5	147	175

Figure 12: Computation of $nullable$, $first$, and $follow$ for some different grammars.

4.2 Using constants before declaration

This is an example described by Farrow in [10] and deals with a language where constants can be defined in terms of other constants and where use of a constant before its declaration is legal as in the following example:

```
a = 2*b + c;
b = 2;
c = d - 1;
d = 4;
```

Farrow shows how a part of an AG for the language can be specified to build a table mapping constant names to their respective values. The specification will be circular. In essence, to build a table of constants and their values you need the value of the expression defining each constant. If an expression defining a constant uses another constant (as in the definitions of `a` and `c` above) you will need to look them up in the table. The table thus depends on the constant values which in turn depend on the table. The only case when cycles will not occur is when no expression defining constants uses other constants, i.e., have the form of the declarations of `b` and `d` above. Had there not been the requirement to allow use of constants before their declaration, the AG could be simplified to avoid cycles. Farrow showed that it is possible to rewrite the AG to be cycle free by introducing complexity involving higher order functions one of which in essence captures the behavior of the fixed-point iterations needed for the evaluation in the cyclic version of the grammar.

Farrow's discussion is based on traditional AGs enhanced with a static evaluation technique for cyclic dependencies mentioned in Section 2. The evaluation will produce the table of constants and their values if the constants are well-defined, i.e., there must be exactly one defining expression for each constant and the definitions themselves must not be cyclic. The table will thus be incomplete if the constants, e.g., are defined as in:

```
a = b + 2;
b = 2*a; // circular definition
```

Using CRAGs it is easy to specify a non-circular attribute grammar for the specification of the constant values. Again a name analysis proves useful linking constant use sites to their declaration sites by a reference attribute `decl` as was described in Section 2.4. The value of a constant can then be modelled as an attribute `val` of its declaration node class. The `val` attribute is specified in terms of the values of the constants used in its defining expression. These values are in turn specified as the value of the `val` attribute at their corresponding declaration sites, using the reference attribute `decl`.

Fig. 13 shows parts of an abstract grammar for a language with integer constants. The specification of the `val` attribute in the `ConstUse` class checks if the constant has been declared. If not, it will be assigned the value `undefined`. The example demonstrates how reference attributes can simplify a grammar as compared to previously suggested solutions.

```
ConstDecl ::= IdDecl Exp { syn Integer val = Exp.val; }
Exp { syn Integer val; }
AddExp : Exp ::= Exp1 Exp2 { val = Exp1.val + Exp2.val; }
...
ConstUse : Exp ::= <ID> {
    syn ConstDecl decl = ...;
    val = decl != null ? decl.val : undefined;
}
IntExp : Exp ::= <INT> { val = <INT>; }
```

Figure 13: CRAG for a language where constants can be used before declaration.

The `val` attribute will be noncircular exactly when Farrow's cyclic specification produces a complete table of constant values, i.e., when each constant has a defining expression and no constant is defined in a circular manner. Should some constants be part of circular definitions like in the example above, this is a programming error that should be caught by the compiler. To use circular attributes is not an option here since there does not, in general, exist any fixed-point solution. For the CRAG above, the evaluator will throw an exception when it discovers the circular dependency. This is, of course, not an acceptable behavior

for a production compiler. An improved CRAG can instead check explicitly for such erroneous circular definitions by introducing an attribute `wellDefined` in class `ConstDecl`. Its value can be specified in a noncyclic manner by a recursive function that builds the set of all constants of which a certain constant is dependent and checks that the constant is not itself in this set. An alternative way is to introduce a circular boolean attribute `wellDefined` in `ConstDecl` and `Exp` with bottom value `false`. The specification of this attribute is straight-forward. A simple integer expression (`IntExp`) is well-defined and a compound expression is well-defined if all its subexpressions are well-defined.

4.3 Live analysis in optimizing compilers

One of the most frequently used examples of cyclic dependencies in AGs is the performance of live analysis for variables. Farrow [10], Jones [15], and Sasaki & Sassa [27] all focus on this example in their papers.

A variable `v` is said to be live on entry to a statement `S` if there is a control flow path from `S` to another point `p` such that `p` uses the value of `v` and `v` is not redefined on the path from `S` to `p`. The goal of an attribute grammar in this context is to specify the sets of live variables on entry to each statement or block in a program.

Farrow and Jones both exemplify with a language with loop-structures like `for`- and `while`-statements. No reference attributes need to be involved here, but the specifications become cyclic for loop-statements. Sasaki & Sassa use a smaller language with only assignment statements, `label`-statements and `goto`-statements. Here remote attributes are used to link `goto` nodes in the AST to their corresponding `label` nodes. Cycles can in their simple example language arise only if a program contains `goto`-statements. Their evaluation technique (or rather the technique to check for convergence) requires that cycles always include remote links. This means that the evaluation process described in [27] would not directly be capable of handling, e.g., a language with structured loop-statements without adding explicit remote links. Also, as has been mentioned before, their links between `gotos` and `labels` in the AST are not ordinary attributes, but need to be provided by a separate phase that takes place before attribute evaluation.

Given the combination of reference attributes and capability of handling cyclic dependencies makes it easy for us to specify a CRAG for live analysis for a language containing ordinary loop-structures as well as `labels` and `goto`-statements. A name analysis links `goto` nodes in the AST to their proper `label` nodes. The rest of the attributes needed to perform a live analysis can be specified following the pattern proposed in earlier papers. In CRAGs, there is no need for an initial phase for computing reference attributes. The reference attributes are evaluated by our system when they are demanded just like any other attributes.

5 Conclusions

In this paper we have presented CRAGs, an extension of traditional AGs with reference attributes and circularly defined attributes. We have developed a general demand-driven evaluation technique for CRAGs, implemented it in Java, and tried it out on several applications, thereby demonstrating the expressiveness of CRAGs and that they are useful for a number of practical problems.

Most language analysis problems include name analysis as a subproblem. It is well known that traditional AGs are not well suited for specifying name analysis, leading to complex awkward specifications. Reference attributes have proved useful to overcome this problem and this paper demonstrates how such name analysis provides a natural basis for further analyses based on circular recursive equations. In Section 4.2 we demonstrated how reference attributes in some cases even remove the need for circular specifications.

Many language analysis problems are inherently circular and need to be computed by iterating to a fixed point. We have demonstrated how CRAGs allow the recursive definitions to be specified directly in the grammar, and the fixed point to be computed by an automatically generated evaluator. The use of reference attributes broadens the potential applications of circular attribute evaluation to a much wider range. The computation of *nullable*, *first*, and *follow*, that we have presented here is representative of a large number of grammar analysis problems that can make use of this technique.

We have compared our demand-driven evaluation algorithm with handwritten imperative code implementing fixed-point iterations, and the results indicate that there is little difference in performance.

Future work includes further improvements of the evaluator. One idea we are looking at is how to isolate strongly connected components by modularizing the grammar. Such modularization is natural to do anyway from a grammar writing perspective, and can probably be used for improving the evaluator performance and for allowing non-monotonic dependencies between components in different modules. We are also looking at techniques for automatically deciding which attributes to cache to provide best performance and memory usage. It would be desirable to let the user decide what attributes to save in the AST nodes and let the tool help to decide when to cache other attributes temporarily to avoid inefficiencies and for check of convergence. One idea could be using a cache like in [27]. We also plan to apply CRAGs to more problem areas. In [9] a formalism for rewriting abstract syntax trees is presented. The formalism, Rewritable Reference Attributed Grammars (ReRAGs), has been implemented in our aspect-oriented compiler tool JastAdd. We plan to merge this extension of JastAdd with our CRAG extension. We believe that there are problem areas where this combination would prove useful.

Acknowledgements

We are grateful to John Boyland and the anonymous reviewers for valuable feedback and helpful comments.

Bibliography

[1] The JavaCC Project, 2007. Available at: http://javacc.dev.java.net.

[2] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[3] J. T. Boyland. *Descriptional Composition of Compiler Components*. PhD thesis, University of California, Berkeley, 1996.

[4] J. T. Boyland. Analyzing direct non-local dependencies in attribute grammars. In *Proceedings of CC'98*, volume 1383 of *LNCS*, pages 31–49. Springer, 1998.

[5] J. T. Boyland. Incremental evaluators for remote attribute grammars. *Electronic Notes in Theoretical Computer Science*, 63(3), 2002.

[6] J. T. Boyland. Personal communication, 2003.

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

[8] A. Cornils and G. Hedin. Tool support for design patterns based on reference attributed grammars. In *Proceedings of WAGA'00, Workshop on Attribute Grammars and Applications*, 2000.

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

[10] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of CC'86*, pages 85–98. ACM Press, 1986.

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

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

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

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

[15] L. G. Jones. Efficient evaluation of circular attribute grammars. *ACM Transactions on Programming Languages and Systems*, 12(3):429–462, 1990.

[16] M. Jourdan. An optimal-time recursive evaluator for attribute grammars. In *International Symposium on Programming*, volume 167 of *LNCS*, pages 167–178. Springer, 1984.

[17] U. Kastens. Ordered Attribute Grammars. *Acta Informatica*, 13(3):229–256, 1980.

[18] T. Katayama. Translations of attribute grammars into procedures. *ACM Transactions on Programming Languages and Systems*, 6(3):345–369, 1984.

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

[20] D. E. Knuth. Semantics of context-free languages (errata). *Mathematical Systems Theory*, 5(1):95–96, 1971.

[21] O. L. Madsen. On defining semantics by means of extended attribute grammars. In *Semantics-Directed Compiler Generation, Proceedings of a Workshop*, volume 94 of *LNCS*, pages 259–299. Springer, 1980.

[22] E. Magnusson. State Diagram Generation using Reference Attributed grammars. LU-CS-TR:2000-219. Technical report, Dept. of Computer Science, Lund University, Sweden, 2000.

[23] E. Magnusson and G. Hedin. Program Visualization using Reference Attributed Grammars. *Nordic Journal of Computing*, 7:67–86, 2000.

[24] P. Persson and G. Hedin. Interactive Execution Time Predictions Using Reference Attributed Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 173–184, Amsterdam, 1999.

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

[26] M. Rodeh and M. Sagiv. Finding circular attributes in attribute grammars. *JACM–Journal of the ACM*, 46(4):556–575, 1999.

[27] A. Sasaki and M. Sassa. Circular Attribute Grammars with Remote Attribute References. In *Proceedings of WAGA'00, Workshop on Attribute Grammars and their Applications*, 2000.

PAPER IV

COLLECTION ATTRIBUTE ALGORITHMS

Abstract

In order to make attribute grammars useful for complicated analysis tasks, a number of extensions to the original Knuth formalism have been suggested. One such extension is the collection attribute mechanism, which allows the value of an attribute to be defined as a combination of contributions from distant nodes in the abstract syntax tree. Another extension that has proven useful is circular attributes, i.e., attributes defined in a circular recursive manner. In this paper we show how collection attributes and the combined formalism, circular collection attributes has been implemented in our declarative meta programming system JastAdd and how they can be used for a variety of applications including devirtualization analysis, metrics and flow analysis. A number of possible evaluation algorithms are discussed and compared for applicability and efficiency. The key design criteria for our algorithms are that they work well with demand evaluation, i.e., defined properties are computed only if they are actually needed for a particular analysis problem. We show that the best algorithms work well on large practical problems including the analysis of large Java programs.

An extended version of: E. Magnusson, T. Ekman, and G. Hedin, "Extending Attribute Grammars with Collection Attributes – Evaluation and Applications.", In the proceedings of SCAM 2007, Seventh IEEE International Working Conference on Source Code Analysis and Manipulation. Best paper award, SCAM 2007.

1 Introduction

Attribute grammars have recently received renewed interest due to the emergence of practical meta programming tools such as JastAdd [1] and Silver [25] that can handle analysis, transformation and compilation of complex programming languages like Java. Main advantages of these systems are that they make use of declarative specifications, allowing high-level concise specifications, and that they support extensibility, for example, allowing advanced analyses to be added modularly to a compiler. Good performance can be achieved, as shown for our own tool, JastAdd, with which we have built an extensible Java compiler that can compile programs in the order of 100 k lines of code and that runs within a factor of three of javac [8, 9].

The practicality of recent attribute-grammar based systems relies on the development of a variety of extensions to the original Knuth style attribute grammars [18]. One such extension allows attributes to reference distant nodes in the abstract syntax tree (AST), and use these references to access attributes in the referenced nodes [4, 12, 13, 16, 21]. We will call this extended formalism *Reference Attributed Grammars* or RAGs. RAGs facilitate specifications of problems where non-local dependencies are common and there are many compiler related tasks where they are essential and provide for concise and clear specifications.

Another useful extension is to allow circular dependencies between instances of attributes [11]. Circular specifications do, under certain circumstances, define well defined attribute grammars in the sense that all semantic rules can be satisfied. Allowing *circular attributes* in some cases makes it possible to more or less directly use underlying recursive definitions. It becomes the responsibility of the attribute evaluator to perform the necessary fixed-point iterations to compute values for attributes involved in circular dependencies. AGs allowing circular definitions are often called CAGs (*Circular Attribute Grammars*). The combined formalism, i.e., AGs incorporating reference attributes as well as circular specifications is called CRAGs (*Circular Reference Attributed Grammars*) [20].

Extensions also include AST *rewriting* where rules are used for specifying transformations of the abstract syntax tree [10], and the related mechanism *higher order attributes* where an attribute can have the strucure of an AST and can itself have attributes [27]. Combined with a *forwarding* technique [26], where equations can be forwarded from one node to another, HAGs allows computations to be expressed on a more suitable model in a manner similar to rewrites. Applications for these extended formalisms include semantic specialization, code optimization and reeingineering of source code.

In this paper we investigate applications and implementation of *collection attributes*, as defined by Boyland [4]. A collection attribute is the declarative specification of a combination of properties of an unbounded number of abstract syntax tree nodes. Simple examples are the set of calls of a procedure, and the set of subclasses of a class. While such combined properties can be computed by ordinary

Knuth-style synthesized and inherited attributes, the use of collection attributes makes their specification much more simple and concise, and opens for more efficient implementations. Furthermore, the use of collection attributes supports the building of extensible analysis tools [24]. In [19] we have presented the collection attribute formalism, presented a number of applications and also discussed possible evaluation algorithms. This paper is an extended and revised version of [19] and includes detailed descriptions of the evaluation algorithms.

Collection attributes are often whole-program properties, i.e., they combine information from, potentially, the whole program. A naive way to implement them is to simply traverse the whole program, find all the contributing AST nodes of the program, and that way compute the combined value. This is potentially very expensive. In this paper we propose a series of evaluation algorithms and compare them, both with regard to applicability and to performance. Our evaluation algorithms are all based on *demand evaluation*, i.e., attributes are not evaluated until their value is demanded by some other computation. This is important for many applications since the set of attributes needed for a computation may depend on the particular program that is analyzed.

Of particular interest is the combination of collection attributes with *circular attributes* [11, 20]. As for ordinary circular attributes, an AG containing circularly defined collection attributes may be considered well defined in that all its semantic rules may be satisfied.

We have implemented collection attributes and the combined formalism circular collection attributes in our system JastAdd and we give several examples of their use, including devirtualization analysis and metrics for Java programs, and flow analysis for grammars.

The rest of this report is structured as follows: In Section 2 we give a motivating example for the introduction of collection attributes. Section 3 discusses the definition of collection attributes and how they are used in JastAdd. Section 4 gives a series of algorithms for evaluation of non-circular collection attributes. Section 5 discusses application examples for non-circular collection attributes. In Section 6 the combined formalism circular collection attributes is introduced and an application example is discussed. Possible evaluation algorithms for circular collection attributes are presented in Section 7. Section 8 is devoted to performance issues for all presented algorithms. Section 9 discusses related work, and Section 10 concludes the paper.

2 Motivation

2.1 The JastAdd system

The JastAdd system [1, 14] allows source code analyses to be built in a concise way as extensions on top of other analyses, typically on top of the name and type analyses that are core parts of a compiler. The analyses are formulated as

attribute grammars (AGs) that include the basic AG mechanisms of synthesized and inherited attributes [18], as well as several extensions, including *reference attributes* [13] that are of key importance to this paper, circular attributes [20] and rewrites [10].

A reference attribute of an abstract syntax tree (AST) node is an attribute that refers to another node in the AST. They are used to bind different parts of the AST together, e.g., to bind a use of a variable to its declaration, a class to its superclass, a call to its method declaration, an expression to a type declaration denoting its type, etc. Using JastAdd, such attributes are typically specified in name and type analysis modules, and a compiler is composed by combining these with a code generation module.

For many source code analysis problems it is useful to reuse the reference attributes computed by the name and type analysis modules. In addition, there is often a need for the reverse information, i.e., the cross references. For example, for a metrics problem, we might be interested in finding all the uses of a particular instance variable declaration, all the calls of a method, or all the subclasses of a class. Cross-reference problems are often whole program problems in the sense that the cross references may be located in any part of the program. For example, a public instance variable declaration can, through qualified use, be used from any other class in the program.

2.2 A motivating example

Later in this paper we will see how cross-reference sets can be specified by means of collection attributes. But first, as a motivation and for comparison, we will look at how they can be specified using ordinary synthesized and inherited attributes[1].

As an example, consider cross references for name bindings. The name analysis module has defined that each `Use` node has an attribute `decl` which refers to the appropriate `Decl` node. We now want to define that each `Decl` node has an attribute `uses` which is a set of cross references, i.e., it contains references to all the `Use` nodes whose `decl` refers to that particular `Decl` node. This can be done by using attributes that in effect traverse the complete AST and collect all the relevant `Use` nodes. The specification is shown in Fig. 1. The `uses` attribute accesses the `collectUses` attribute of the root. The `collectUses` attribute is defined for all AST nodes (in the superclass `ASTNode`), and by default collects the uses of its children. This in effect results in a traversal of the complete AST. For `Use` nodes, the `collectUses` attribute in addition adds a reference to that `Use` node, if appropriate, i.e., if its `decl` refers to the `Decl` in question.

The JastAdd system evaluates attributes through *demand evaluation*. This means that the value of an attribute is not computed until its value is needed. This is important for efficiency because there may be many attributes defined whose

[1] A synthesized attribute is defined by an equation in the same AST node. An inherited attribute is defined by an equation in an ancestor node.

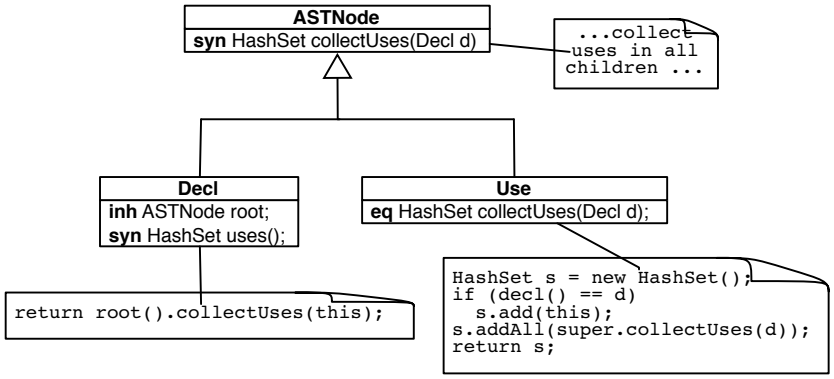

Figure 1: Finding all uses of a declaration.

values are not needed for a particular application. For our example we can note that there is one instance of the `uses` attribute for each declaration in the analyzed program. It might be the case that we are interested in a `uses` attribute instance only if some condition holds. For example, if the declaration has a particular name.

In analyzing the specification of the `uses` attribute above, we can notice some drawbacks. First and foremost, the evaluation is inefficient if several instances of the attribute are demanded since a complete tree traversal is performed for every instance. Furthermore, the user has to explicitly express the tree traversal using auxiliary attributes like `collectUses`. In the next section, we will see how both these drawbacks can be overcome through the use of collection attributes.

3 Collection Attributes

3.1 Definitions

The value of an ordinary synthesized attribute of an AST node *n* is defined locally by an equation in node *n*. In contrast, the value of a *collection attribute*, as defined by Boyland [4], is defined through a number of partial definitions, located in arbitrary nodes in the AST. More precisely, the value is defined as a combination of an *initial value* and zero or more applications of a *combination operation*. The collection attribute declaration contains the initial value and the combination operation. The partial definitions, in the form of applications of the combination operation, can be located in arbitrary AST nodes. Following the declarative paradigm of AGs, the order of specification is irrelevant. Therefore, the combination operation must be such that the order of application does not affect the resulting value of the

collection attribute. Typical examples of collection attribute types are *sets*, with the empty set as the initial value and *add element* as the combination operation; *booleans*, with false and *or*; and *integers*, with zero and +.

In order to facilitate the description of evaluation algorithms we introduce some additional terms. A node containing a partial definition for some collection attribute `c` is said to be a *contributor* to its final value. Alternatively, we say that the node *contributes* to the value of `c`. The value of the partial definition is said to be its *contribution*.

3.2 Motivating example revisited

Consider again the problem of finding all uses of a declaration, as was described in Section 2. After introducing collection attributes, we can replace the synthesized attribute `uses` with a collection attribute as shown in Fig. 2. It has the initial value `new HashSet()` and the combination operator `add`. The nodes of type `Use` are contributors of `Decl.uses` and their respective contribution is `this`, i.e., a reference to the `Use` node. The reference attribute `decl` (that was defined in the name analysis module) points out the appropriate `Decl` node to contribute to.

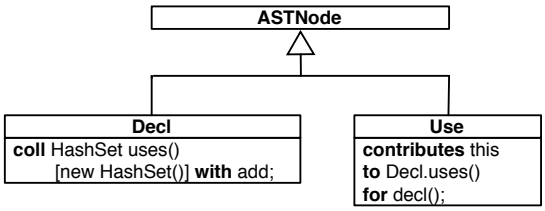

Figure 2: Defining uses with a collection

This definition is much more concise than the one using ordinary attributes. We can also notice that the combination operator `add` is an ordinary Java method that updates the state of the `HashSet` object. This is fine, since the value of `uses` will not be used until its final value is computed. In contrast, the solution shown in Section 2 exposes all the partial collections as attributes, and therefore needs to represent them as separate objects. This difference will contribute to better efficiency for the collection attribute solution.

3.3 JastAdd collection attribute syntax

The JastAdd syntax for declaring a collection attribute `c` of type `T` in a node class `N1` is:

```
coll T N1.c() '[' initial ']' with op;
```

The declaration of the attribute `c` is an *intertype declaration* in that it allows the declaration to be expressed in a module textually separate from class `N1`, similar to intertype declarations of methods and fields in, for example, AspectJ [17].

Alternatively, the following syntax can be used:

```
class N1{
  coll T c() '[' initial ']' with op;
}
```

This syntax is convenient to use when there are several intertype declarations for the same class.

The expression `initial` is the initial value of the collection attribute, before applying the contributions, and thereby also the final value in the case of zero contributions. The `op` should be a Java method for class `T` that serves as the combination operator and updates the `T` object by adding a contribution. The method `op` should be void and have a single parameter of the same type as the contributions[2]. Given a set of contributions E, the final value of `c` is computed as follows (pseudo code):

```
T c = initial;
for all e ∈ E do
  c.op(e);
end for
```

The uses example in Fig. 2 illustrated a very simple example of declaring contributions. In more complex cases it can be desirable to express conditional contributions, and contributions for a set of collection attributes, not just for a single attribute. Below, the general syntax for declaring contributions from a node `N2` to collection attributes `c` in `N1` nodes is shown.

```
N2 contributes
contr1 [when cond1],
contr2 [when cond2],
...
contrN [when condN]
to N1.c()
for [each] ref();
```

The contributions `contr1`, `contr2`, etc., should be expressions that have the same type as the parameter of the combination operator of `c`. The semantics of a specification with when-clauses is that all contributions for which the corresponding condition holds are valid. The expression `ref` should be a reference to an `N1` object, or, if the optional keyword `each` is used, `ref` should be a *set* of references to `N1` objects. In the latter case, the contribution is added to the collection attribute `c` of *all* those `N1` objects.

[2] If collections of primitive types like int and boolean are desired, they currently have to be implemented by wrapper classes.

Below, an example of using `for each` in a contribution is given. The class `MethodDecl` has a collection attribute `calls` (defined elsewhere) which contains references to all methods called inside its body. We now want to define a cross-referencing collection, `callers`, that should contain references to all methods that call the `MethodDecl`. This is accomplished concisely by letting the `MethodDecl` contribute itself to all the `caller` attributes of each of the members in its `calls` attribute.

```
coll HashSet MethodDecl.callers() = new HashSet() with add;

MethodDecl contributes this
to MethodDecl.callers()
for each calls();
```

4 Evaluation of non-circular Collection Attributes

4.1 Attribute evaluation in JastAdd

In the demand driven evaluation technique used in JastAdd for non-collection attributes, an attribute instance is not computed until its value is needed. If the value depends on other attribute instances, these instances are demanded and evaluated as well. The evaluator code for ordinary synthesized and inherited attributes is realized by translating their declarations and equations into Java methods, as described in more detail in [14]. For efficiency reasons, a caching technique is also available. Circular attributes are always cached since the iterative technique used for their evaluation requires values from a previous iteration to be cached for convergence check. When the iterative process is over, these cached values contain the final values of all attributes involved in the cycle. Collection attributes are also always cached since their computation involves traversing the whole AST, and is thus inherently expensive.

When implementing collection attributes, we want to keep with the demand-driven approach, so that the evaluation of the attributes demanded in a particular application is not slowed down by the existence of attributes that are not demanded.

This section contains, descriptions of a number of evaluation techniques for collection attributes and their applicability while performance issues are treated in a later section. Descriptions and code examples for all alternatives assume that a collection attribute has been declared and specified as follows:

```
coll T A.s() [init_value()] with op;

B contributes
f(..)
to A.s()
for ref();
```

In order to facilitate the discussion of applicability we introduce some additional terms:

- All instances of node classes which specify contributions to a collection attribute will be called *potential contributors* to an instance of the attribute. Thus, all instances of class `B` are potential contributors to any instance of the collection attribute `s` in the example above.
- Let `ref` denote the reference attribute used in contributing classes to express for which node a contribution is intended. The *actual contributors* to a certain instance `s1` of a collection attribute `s` is the subset of its potential contributors, for which `ref` points out the node site of `s1`.
- A contribution from an actual contributor is a *valid contribution* if it is either unconditional or conditional with an attached condition which evaluates to true.

4.2 Naive evaluation

A simple way to implement a collection attribute is to represent the collection attribute by a method that traverses the complete AST, finds the appropriate contributors, and returns the final value, i.e., the combination of the contributions. We refer to this evaluation scheme as the *naive evaluation algorithm*. This algorithm has the advantage that it is purely demand driven: if a single collection attribute instance is demanded, there is no extra work involving the evaluation of other instances. But if several instances are demanded, the tree will be traversed over and over again, leading to overall inefficiency.

The essential parts of the evaluator code for the naive technique is shown in Fig. 3. (Code needed for interaction with rewrites and certain optimization issues for circular evaluation has been omitted.) The method `s()` first checks if `s` has already been computed. If not, a method in the root node is called to compute its value with the node of the demanded attribute instance as a parameter. `root` is assumed to be an ordinary reference attribute specified to reference the root node of the AST. Node class `ASTNode` is supposed to be the superclass for all node classes of the AST. In `ASTNode` the method for computing the attribute `s` of `A` is defined to simply call itself for all children. In the contributing class `B` this method is overridden. Here it is checked if the reference attribute `ref` references the same node as the parameter. If this is the case, the contribution is combined with the current value of the `s_value`. Then the corresponding method in the superclass (in this case `ASTNode`) is called to continue the tree traversal.

If contributions are conditional, a check of these conditions must be made in contributing nodes. If, for example, the contribution given to `A.s` from class `B` has an attached condition, then the generated code for the method `A_s_compute` in `B` would also check this condition before combining the contribution with `s_value`.

Should the `for each` construct be used to specify contributions, then the code of `A_s_compute` will include an iteration over the set `ref`. For each element it will then be checked if it points out the demanded instance, and if this is the case the combination will be performed.

```
class A {
  boolean s_computed = false;
  T s_value = init_value();

  public T s() {
    if (s_computed)
      return s_value;
    root().A_s_compute(this);
    s_computed = true;
    return s_value;
  }
}
class ASTNode {
  void A_s_compute(A n) {
    for(int i = 0; i < getNumChild(); i++)
      getChild(i).A_s_compute(n);
  }
}
class B {
  void A_s_compute(A n) {
    if (ref() == n) {
      n.s_value.op(f(...));
    }
    super.A_s_compute(n);
  }
}
```

Figure 3: Naive evaluation of collection attribute `s` in class `A`.

Applicability of the naive technique

For a discussion of the applicability of evaluation algorithms some basic facts need to be established. Firstly, we can state that in order to compute the value of a collection attribute instance we need the values of the referencing attributes in *all* its potential contributors. The reason is that all these reference attributes must be evaluated in order to find the actual contributors. Secondly, we also need the values of conditions attached to contributions in the actual contributors of the collection attribute instance. Thirdly, we need the values of the valid contributions. Any instance of a collection attribute is thus dependent on the reference attributes in all potential contributors, all conditions in actual contributors and their valid contributions. We will call this type of dependency *inherent* since it exist regardless of which algorithm is used for evaluation.

The inherent dependency on the reference attributes is of particular interest, since it introduces circularities whenever any of the reference attributes are dependent on an instance of the collection attribute. To see this, assume that the value of an instance `s1` of a collection attribute `s` is needed. The instance `s1` depends on all reference attributes in potential contributors. If one of them, say `ref1`, is dependent on an instance of the collection attribute, say `s2`, then we have a cycle since `s2` depends on all references, among them `ref1`. Using the code presented above, this type of dependencies causes looping behavior. Evaluation of `ref1` causes `s2` to be demanded and thereby a new tree traversal is triggered. During this traversal all references, among them `ref1`, will be needed again and therefore `s2` is demanded again. In this case, `ref1` is the first attribute instance to be revisited during the evaluation. The reference attributes are normally ordinary synthesized or inherited attributes. The evaluator code generated for such attributes are capable of detecting looping behavior as described in [20].

The inherent dependency on conditions and contributions does not create cycles in the same way. If, for example, a condition is dependent on another instance of the collection attribute, it will simply trigger the evaluation of the other collection attribute instance and then continue with the evaluation of the first one. Only if dependencies are actually circular, i.e., an instance is dependent on itself, will there be problems. Such circular dependencies can be detected dynamically using the same technique as for ordinary non-circular attributes as described in [20]. In essence, a boolean Java attribute `s_visited` is introduced in class `A` and used in the method `s()` as outlined in Fig. 4.

When the naive technique is used and failure is reported by the evaluator as in Fig. 4, it is necessary to resort to other algorithms capable of handling collection attributes involved in circular dependencies. Such algorithms will be discussed in a later section.

4.3 One-phase joint evaluation

To obtain better overall efficiency, it is possible to compute *all* instances of a given collection attribute when the first instance is demanded. One traversal of the tree is sufficient to find all contributing nodes to any instance of the attribute and to perform the proper computations for combining contributions. We call this technique *one-phase joint evaluation*. This scheme deviates from the demand-driven technique. If only a single attribute instance is actually demanded, it will be less efficient than the naive algorithm. But if more instances are demanded, it will quickly become much more efficient. An outline of the evaluator code is given in Fig. 5. The evaluator method for `s` in class `A` is no longer parameterized. If the attribute is not already computed it calls a method in the root node. For the root node, `Program`, this method checks whether the tree has already been traversed (using a flag `A_s_computed`). If not, a tree traversal is triggered and the flag is set to indicate that all instances of `s` have been computed. As before, in the

```
class A {
  boolean s_computed = false;
  boolean s_visited = false; // new
  T s_value = init_value();

  public T s() {
    if (s_computed)
      return s_value;
    if (s_visited)
      throw new RuntimeException("..."); // new
    s_visited = true; //new
    root().A_s_compute(this);
    s_visited = false;  // new
    s_computed = true;
    return s_value;
  }
}
```

Figure 4: Detecting circularities in the naive technique.

naive technique, the computing method is overridden in the contributing class `B`. Here, the contribution is combined with the current value of `s_value` in the node referenced by `ref`.

As for the naive technique, the generated code for `A_s_compute` is modified if conditions are attached to contributions or if the `for each` construct is used.

Applicability of the one-phase technique

A shortcoming of the one-phase algorithm is that it is less general than the naive technique. As described in connection with the naive technique, a collection attribute instance is inherently dependent on the reference attributes in all potential contributors, on conditions in actual contributors and on their valid contributions. For the one-phase technique there are also additional *algorithmic dependencies* in the sense that the algorithm creates dependencies by enforcing a certain order of computations. Before the value of any instance of a collection attribute is returned the algorithm enforces not only *all* references but also *all* conditions for potential contributors and *all* contributions valid for any instance to be evaluated. Any instance of collection attribute thus becomes dependent on all these entities. With the same reasoning as for the naive technique we can therefore deduce that the algorithm will have a looping behavior whenever any reference attribute or any condition for a potential contributor or any of their valid contributions is dependent on any instance of the collection attribute under computation. The dependency will then trigger a new tree traversal, during which the same dependency will be encountered, and the algorithm will end up in a loop.

```
class A {
  boolean s_computed = false;
  T s_value = init_value();

  public T s() {
    if (s_computed)
      return s_value;
    root().A_s_compute();
    s_computed = true;
    return s_value;
  }
}
class Program {
  boolean A_s_computed = false;

  void A_s_compute() {
    if (A_s_computed) return;
    super.A_s_compute();
    A_s_computed = true;
  }
}
class ASTNode {
  void A_s_compute() {
    for(int i = 0; i < getNumChild(); i++)
      getChild(i).A_s_compute();
  }
}
class B {
  void A_s_compute() {
    ref().s_value.op(f(...));
    super.A_s_compute();
  }
}
```

Figure 5: One-phase joint evaluation of collection attribute `s` in class `A`.

The evaluator code for the one-phase technique can be modified to detect looping behavior and report failure by simply adding a boolean instance variable to the root class and use it in the method starting tree traversal as shown in Fig. 6.

```
class Program {
  boolean A_s_computed = false;
  boolean A_s_computing = false; // new

  void A_s_compute() {
    if (A_s_computed) return;
    if (A_s_computing)
      throw new RuntimeException("..."); //new
    A_s_computing = true; //new
    super.A_s.compute();
    A_s_computing = false; //new
    A_s_computed = true;
  }
}
```

Figure 6: Detecting dependencies leading to looping behavior in the one-phase technique.

4.4 Two-phase joint evaluation

In order to avoid the inefficiency of the naive technique and to avoid the shortcomings of the one-phase technique, we propose an alternative technique, *two-phase joint evaluation*.

Survey phase The first time any instance of the `s` attribute is demanded, a traversal of the AST is triggered. During this traversal, contributors to *all* instances of `s` are collected into *contributor sets*, one set for each instance of `s`, and stored in an auxiliary attribute `s_contributors`, in class `A`. A flag is set to indicate that this *survey phase* has been performed.

Combination phase To compute the value of an instance of `s`, the flag is first checked to see if the survey phase has already been run. If not, this phase is performed first. Then the *combination phase* is run: the attribute value is computed by iterating over the elements in the corresponding `s_contributors` set, and combining the contributions using the combination operation. The final value of the `s` instance is cached so that subsequent demands for it can return the value directly.

For conditions attached to contributions we have two variants of the two-phase algorithm: to evaluate the conditions during the survey phase, *early condition evaluation*, or to postpone these computations until the combination phase, *late condition evaluation*. In the rest of this paper we will call the first variant 2Ph-EC and

the second variant 2Ph-LC. If conditions are evaluated during the survey phase, and there is more than one when-clause, they will have to be checked again during the combination phase in order to find the valid contributions.

An outline of the evaluator code for the two-phase scheme is given in Fig. 7. As before, `root` is assumed to be an ordinary reference attribute referencing the root node of the AST and `Program` is assumed to be the class of the root node. As for the one-phase technique, it is first checked whether the demanded instance has already been computed. If not, a method in the root node is called to perform the survey phase. If the survey phase has already taken place, this method returns immediately. Otherwise, a tree traversal is triggered collecting contributors in appropriate sets. The evaluator method, `s()`, then combines its contributions. This is realized by calling a method `contributeTo` for each member in the contributor set. This method is called with the current non-final value of the instance under computation as a parameter. In the contribution class, in this case `B`, `contributeTo` combines its contribution with the current value of `s`.

If conditions are attached to the contributions, then some additional code is added. For the 2Ph-EC variant the `collect_contributors_A_s` method is modified to check conditions before adding a node to the appropriate contributor set. For both variants conditions must also be checked in the `contributeTo_A_s` method in the contributing class before combining a contribution with the current, non-final value of the attribute under computation.

If the `for each` construct is used for expressing contributions, then the generated code of `collect_contributors_A_s` method will include an iteration over the set `ref`. The `add` operation will be applied to contributor sets of each node referenced by a member in this set.

Applicability of the two-phase techniques

Late condition evaluation. With respect to applicability, the 2Ph-LC technique is equivalent to the naive technique. I. e., it will fail (loop) for a collection attribute that is circularly defined but succeed otherwise. To see this, we simply note that the only order of computation enforced by this technique is that it evaluates references in all potential contributors (in the survey phase) before any collection attribute instance. This does not introduce any additional algorithmic dependencies since any collection attribute instance is inherently dependent on these reference attributes.

If the reference attributes introduce dependencies between instances of the collection attribute, then the algorithm outlined in Fig. 7 will end up in a loop. Assume that an instance `s1` is demanded. During the survey phase all reference attributes are evaluated. Assume that one of them, say `ref1`, depends on another instance `s2`. `s2` is then demanded and the survey phase is re-entered. During this phase, the reference `ref1` will be demanded again. Since it depends on `s2` there will be a looping behavior. The code can be

```
class A {
  boolean s_computed = false;
  T s_value;
  HashSet s_contributors = new HashSet();

  public T s() {
    if(s_computed)
       return s_value;
    root().collect_contributors_A_s();
    s_value = init_value;
    for(Iterator iter = s_contributors.iterator(); iter.hasNext(); ) {
      ASTNode contributor = (ASTNode)iter.next();
      contributor.contributeTo_A_s(s_value);
    }
    return s_value;
  }
}
class Program {
  boolean has_collected_contributors_A_s = false;
  void collect_contributors_A_s() {
    if (has_collected_contributors_A_s) return;
    super.collect_contributors_A_s();
    has_collected_contributors_A_s = true;
  }
}
class ASTNode {
  void collect_contributors_A_s() {
    for(int i = 0; i < getNumChild(); i++)
      getChild(i).collect_contributors_A_s();
  }
  void contributeTo_A_s(T c) {
  }
}
class B {
  void collect_contributors_A_s() {
    ref().s_contributors.add(this);
    super.collect_contributors_A_s();
  }
  void contributeTo_A_s(T c) {
    c.op(f(...));
  }
}
```

Figure 7: Two-phase evaluation of collection attribute `s` in class `A`.

exended to detect looping behavior in the same way as described before for the naive technique.

Looping behavior caused by the reference attributes can also be captured earlier, as soon as the survey phase is re-entered. This is accomplished by extending the code of the survey phase as shown in Fig. 8.

```
class Program {
  boolean has_collected_contributors_A_s = false;
  boolean collecting_contributors_A_s = false;  // new
  void collect_contributors_A_s() {
    if (has_collected_contributors_A_s) return;
    if (collecting_contributors_A_s)
      throw new RuntimeException("..."); // new
    collecting_contributors_A_s = true; // new
    super.collect_contributors_A_s();
    collecting_contributors_A_s = false; //new
    has_collected_contributors_A_s = true;
  }
}
```

Figure 8: Detecting looping behavior during the survey phase of the two-phase technique.

When the 2Ph-LC technique reports failure, an algorithm capable of handling circular dependencies must instead be used.

Early condition evaluation. The 2Ph-EC algorithm is less general than the 2Ph-LC variant in that it will fail to evaluate certain collection attributes even if they are not circularly defined. To see this, we observe that references as well as conditions attached to contributions are evaluated for all potential contributors during the first (survey) phase. This adds algorithmic dependencies of collection attribute instances on the conditions of *all* potential contributors. There will therefore be looping behavior whenever any of these conditions are dependent on any instance of the collection attribute under computation. Assume, for example, that an instance `s1` is demanded. During the survey phase all conditions are evaluated. Assume that one of them depends on another instance `s2` of the collection attribute. `s2` is then demanded and the survey phase is re-entered. During this phase, the same conditions will be evaluated and therefore `s2` will be demanded again. Hence, such collection attributes cannot be evaluated by the 2Ph-EC technique.

The code can be modified to detect looping behavior in the same manner as for the 2Ph-LC variant.

Failure during the combination phase of the 2Ph-EC technique happens only when dependencies are actually circular. In this case an algorithm capable of handling circular dependencies must be used. Failure during the survey phase can either be caused by algorithmic dependencies (introduced by the conditions) or inherent dependencies (introduced by the reference attributes). In the former case, the 2Ph-LC algorithm can be used instead. If, on the other hand, the survey phase fails because of inherent dependencies through the reference attributes, then dependencies are actually circular and an algorithm for circular collection attributes must be used.

4.5 Additional variants

Grouped joint evaluation The two-phase techniques and the one-phase technique deviate from pure demand evaluation. When, for the first time, an instance is demanded they both perform computations that might not be needed in the future. In the two-phase techniques the survey phase prepares for later demands of other instances by collecting contributor sets. The one-phase technique fully computes all instances. Efficiency can, in some cases, be improved by grouping two or more collection attributes. In the two-phase techniques the survey phase can be modified to collect contributors for all instances of all attributes in the same group in one tree traversal. The one-phase technique can be modified to fully compute all instances of the grouped attributes when for the first time an instance of any one of them is demanded. Detailed code is omitted.

In JastAdd the user can require grouped evaluation of collection attributes by annotating their declarations as in the following example:

```
coll @CollectionGroup("GroupOne") Set A.s() with add;
coll @CollectionGroup("GroupOne") Set B.t() with addAll;
```

`CollectionGroup(...)` is used to indicate that grouped evaluation is required. Groups are named by the string parameter and attributes annotated with the same name belong to the same group.

Grouped evaluation combined with the one-phase technique fails if there are dependencies between any instances of attributes in the group caused by references, conditions or contributions.

Grouped evaluation combined with the 2Ph-LC technique works well as long as the references used in potential contributors for attributes in the group are not dependent on any instance of an attribute in the group. If instead a combination with the 2Ph-EC technique is used, failure occurs when these references or conditions in potential contributors for attributes in the group are dependent on any instance in the group. In both cases the survey phase is re-entered which causes looping behavior. Failure can be detected

and reported by the evaluator in a similar manner as for the un-grouped techniques described above.

Pruned traversal Through a simple analysis of the abstract grammar, it can be determined which AST types can be derived from a given AST type `T`. If none of these AST types contain contributions for a given collection attribute `C`, then the AST traversal for `C` can be pruned at `T` nodes, thereby speeding up the traversal. However, for Java, such prunings cannot be expected to be other than marginal due to the very recursive structure of the language. For example, a Java expression can derive an anonymous class, which means that most AST types can be derived from expressions.

4.6 Summary

We have presented a series of algorithms for evaluation of non-circular collection attributes differing with respect to applicability. The following algorithms have been implemented in JastAdd: Naive, 2Ph-LC, 2Ph-EC and 1Ph-Joint. We have also implemented grouped evaluation in combination with the 2Ph and and the 1Ph-Joint techniques. Pruned traversal is not implemented.

With respect to applicability the Naive and the 2Ph-LC algorithms are equivalent. They work as long as there are no circular dependencies between collection attribute instances. The 2Ph-EC is less general in that it fails whenever the 2Ph-LC algorithm fails and also if there are dependencies between instances of the collection attribute introduced by the conditions attached to contributions in potential contributors. The 1Ph-Joint technique is the least general as it fails whenever the 2Ph-EC fails and also if valid contributions introduce interdependencies.

Whenever the 2Ph-LC technique fails for a certain collection attribute `C` then the grouped alternative of the algorithm will fail for a group containing `C`. This happens when any of the references used to point out nodes for contributions is dependent on another instance of the same attribute. The grouped alternative will also fail if such a reference is dependent on an instance of another collection attribute in the group. Therefore the 2Ph-LC is more general than the 2Ph-LC-Grouped. Using similar arguments we can also deduce that the 2Ph-EC is more general than the 2Ph-EC-Grouped and that the 1Ph-Joint is more general than the 1Ph-Joint-Grouped.

In JastAdd, the user can for each individual attribute, choose which algorithm to use. This is done through annotations of declarations for collection attributes. We have chosen the 2Ph-LC algorithm as the default method. It will be used for collection attributes for which declarations are not annotated. The reason for choosing this algorithm is that it is general, working as long as there are no circular dependencies. It is also more efficient than the equally general naive algorithm, which will be shown in Section 8.

5 Application Examples for non-circular Collection Attributes

In this section we will discuss some application examples for which part of the specification can be naturally expressed using non-circular collection attributes.

5.1 Devirtualization

For object-oriented languages it is possible to improve execution speed by applying devirtualization techniques. The aim is to determine statically which virtual method calls can be replaced by static method calls. There are many different techniques for devirtualization based on analyzing the class hierarchy and the call graph of the program, e.g., [7, 23].

The simplest condition for devirtualization is that a class has no subclasses. If for a methodcall `a.m()` the declared type of `a` is a class `A` with this property, then `m()` can be devirtualized. In Fig. 9 this simple criterion is checked. It is assumed that the synthesized attribute `superClasses` is specified as the set of all superclasses for a class declaration. The cross-referencing collection attribute `subClasses` models the set of subclasses. The attribute `ZSdevirt` (Zero Subclasses devirtualization), declared in class `VirtualMethodAccess` uses the collection attribute to check if the target of the method access is a method belonging to a class with no subclasses. The specification of this attribute uses values of some other attributes: `isQualified`, `qualifier` and `hostType` in order to establish the formal type of the receiver of the method access.

It is also possible to devirtualize a method access `a.m()` if `A` has subclasses but `m()` is not overridden in any of them. In Fig. 10 part of a devirtualization analysis based on this criteria is specified. It is assumed that the synthesized attribute `overrides` for a method declaration is specified to contain references to all method declarations that it overrides. A collection attribute `overriders` is then simple to model as a collection attribute cross-referencing `overrides`. In `VirtualMethodAccess` the critera for devirtualization is modelled as a synthesized attribute `NOMdevirt` (No Overriding Methods devirtualization) checking if the target of the access is overridden in subclasses below `A`.

Devirtualization can be improved if reachability is taken into account as in the RTA (Rapid Type Analysis) algorithm [2]. This algorithm uses information about global class instantiation and class hierarchy. In order to decide whether a class is instantiated we need to find `new` expressions inside reachable methods. The property for a method of being reachable can be defined in a recursive manner: the `main` method is reachable and other methods are reachable if any of their callers are reachable. This can be modelled as a circular attribute `reachable` using an auxiliary attribute `callers` for its specification. The `callers` attribute is naturally expressed as a collection attribute cross-referencing an ordinary attribute

```
class ClassDecl {
  syn HashSet superClasses() = ...;
  coll HashSet subClasses() [new HashSet()] with add;
}

ClassDecl contributes  this
to ClassDecl.subClasses()
for each superClasses();

class VirtualMethodAccess {
  syn boolean ZSdevirt() {
    TypeDecl F = isQualified()
          ? qualifier().type()
          : hostType(); // Formal type of receiver
    if (F instanceof ClassDecl)
      return ((ClassDecl) F).subclasses().size()==0;
    else return false;
  }
}
```

Figure 9: Checking the zero subclasses criterion for devirtualization

```
class MethodDecl {
  syn HashSet overrides() = ...;
  coll HashSet overriders() [new HashSet()] with add;
}

MethodDecl contributes  this
to MethodDecl.overriders()
for each overrides();

class VirtualMethodAccess {
  syn boolean NOMdevirt() {
    TypeDecl F = isQualified()
      ? qualifier().type()
      : hostType(); // Formal type of receiver
    for(Iterator itr=decl().overriders().iterator(); itr.hasNext();) {
      MethodDecl mo = (MethodDecl) itr.next();
      if (mo.hostType().instanceOf(F))
        return false;
    }
    return true;
  }
}
```

Figure 10: Checking the no overriding methods criterion for devirtualization.

`calls` modelling the set of methods called from inside a method body. This is shown in Fig. 11.

```
class MethodDecl {
  syn HashSet calls() = ...;
  coll HashSet MethodDecl.callers [new HashSet()] with add;

  syn boolean reachable() circular [false] {
    if (name().equals("main")) return true;
    for (Iterator itr = callers().iterator(); itr.hasNext();) {
      MethodDecl m = (MethodDecl) itr.next();
      if (m.reachable()) return true;
    }
    return false;
  }
}

MethodDecl contributes this
to MethodDecl.callers()
for each calls();
```

Figure 11: Collection attribute used in reachability computation.

5.2 Metrics

To evaluate the use of collection attributes in a real-life application we implemented Chidamber and Kemerer's set of object-oriented metrics [6]. These metrics include structural properties such as the height of the inheritance tree and the number of subclasses, but also more global properties such as the coupling between classes. Internal properties such as the lack of cohesion of methods within a class and the number of weighted methods per class are also computed.

The metrics are implemented as a modular extension to our Java 1.4 checker and consists of 7 collection attributes, 17 contribution declarations, and 12 synthesized utility attributes. The entire specification, including code for printing the metrics for each type, is 165 lines of code excluding comments, and is available on the JastAdd web site [1]. We compare our implementation to the program *ckjm* [22] which provides an alternative implementation for the same set of metrics but that is using a set of visitors on top of the Byte Code Engineering Library (BCEL). That implementation is also completely modular but more than twice as large, being 380 lines of code without comments. The collection attribute based implementation has a number of improvements compared to the visitor based solution. The implementation is not as tangled, i.e., each attribute computes a single metric rather than interleaving multiple ones within a visitor. Another improvement is that each metric is implemented by a set of equations in a single module rather than being scattered across multiple visitors. The declarative attributes also

alleviates the programmer from the manual scheduling of visitor passes, and the temporary storage of intermediate state.

6 Circular Collection Attributes

In [20] we showed how the combined formalism CRAG (Circular Reference Attributed Grammar) supporting reference attributes and circularly defined attributes enhances the expressiveness of AGs. Having introduced collection attributes it is therefore natural also to explore the possibility to handle collection attributes involved in circular dependencies. In this section we will discuss this extension and illustrate with an example of its use.

6.1 Circular specifications in Attribute Grammars

In traditional AGs circular dependencies are considered an error. The requirement for non-circularity is, however, a sufficient but not necessary condition to guarantee that the AG is well defined, i.e., that all semantic rules can be satisfied. As Farrow showed [11], it actually suffices that all attribute instances involved in circular dependencies have a fixed point that can be computed with a finite number of iterations. Farrow showed that one way to ensure that this is possible for ordinary synthesized and inherited attributes is by requiring that the domains of the circularly dependent attributes can be arranged in a lattice of finite height and that all semantic functions are monotonic.

The requirement concerning lattices can be directly carried over to collection attributes. The requirement concerning the semantic functions of ordinary attributes corresponds to a requirement of monotonicity for the composed application of the combination operation to the contributions. If these requirements are met, it is possible to evaluate collection attribute instances involved in cycles by applying an iterative technique until a fixed point is reached.

In JastAdd, the user can state that a collection attribute might be involved in cycles by simply adding the keyword `circular` to its declaration. The default-value given in the declaration will then serve as the start value for iterations in the generated evaluator code. Contributions to circular collection attributes are expressed in the same manner as for non-circular ones.

6.2 Application Example

In this subsection we will give an example of an application involving recursive circular specifications. First, a solution using ordinary circular attribute will be presented. Then, we show how this specification becomes shorter and much more concise when circular collection attributes are used.

Consider the problem of computing the *nullable* property and the *first* and *follow* sets for nonterminals of a context free grammar. As is shown in [20] the

definitions of all these are recursive with possible circularities. For example, a nonterminal `N` is nullable if the empty string can be derived from it. Therefore it is nullable if the empty string can be derived from the right-hand side of at least one of its productions. A necessary condition is thus that all the nonterminals appearing on the right-hand side of one of its productions are nullable. The definition is recursive with possible circularities as `N` might be one of these nonterminals or might be derivable from one of them.

The abstract grammar of a simple language for context-free grammars (CFGs) is shown in Fig. 12. A context-free grammar (`CFG`) is here defined as a list of `Rules`. A `Rule` corresponds to all productions for a certain nonterminal symbol `NDecl`. `Prod` models the right-hand side of an individual production. Uses of nonterminals, occurring in production right-hand sides, is modelled by `NUse`. As shown in Fig. 13 the concrete grammar uses the symbol `|` to separate alternative right-hand sides and the character ϵ to denote the empty string. An example of a

```
CFG ::= Rule*;
Rule ::= NDecl Prod*;
Prod ::= Symbol*;
abstract Symbol;
Terminal: Symbol ::= <TERMINAL>;
NUse: Symbol ::= <ID>;
NDecl ::= <ID>;
```

Figure 12: Abstract grammar for a simple language for context-free grammars.

CFG belonging to this language is found in Fig. 13, where capital letters are used for nonterminals. The AST of this grammar is shown in Fig. 14.

```
X ::= Y | a
Y ::= b | ϵ
Z ::= XYZ | c
```

Figure 13: Example of a grammar belonging to the language of Fig. 12.

Specification using ordinary circular attributes

As shown in [20] the *nullable* property as well as the *first* and *follow* sets for a context-free grammar can be modelled as circular attributes in an AG. For *nullable* and *first* this is straightforward, more or less directly translating their recursive definitions into equations. These specifications will not be shown here. For *follow*, the specification is more complicated. The reason is that all places where a

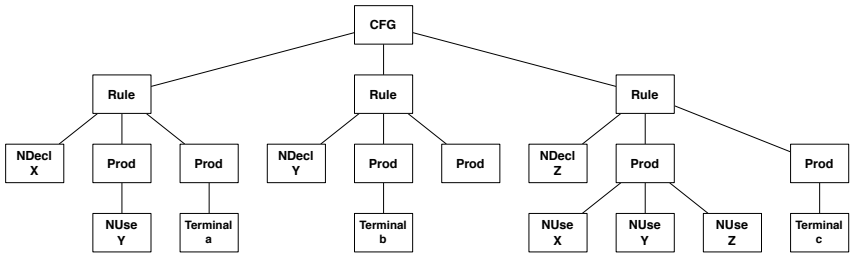

Figure 14: AST of the grammar in Fig. 13.

nonterminal is used in productions contribute to its *follow* set and these places are distributed all over the grammar.

The specification of *follow* presented here, using ordinary circular attributes, assumes that the property *nullable* and the set *first* have been modelled and specified as circular attributes `nullable` and `first` for node class `NDecl` as was shown in [20].

A specification of a circular attribute `follow` declared in class `NDecl` relies on retrieving all places in the grammar where a certain nonterminal is used. For this reason an auxiliary attribute `occurrences(String name)` is declared in class `NDecl`. It models the set of all `NUse` instances where the nonterminal is used. Its specification is shown in Fig. 15. The specification relies on the value of the auxiliary set-valued attribute `findUses(String name)` for the root node. This auxiliary attribute is declared in `ASTNode` where its default specification performs a tree traversal collecting uses from sub-trees. This default behavior is overridden in class `NUse`. Here it is checked if the parameter matches the name of the `NUse` instance. If this is the case, the instance is added to the set before the tree traversal continues. For example, when all use sites for the nonterminal `Y` of example CFG in Fig. 13 are needed, the value of the attribute `findUses("Y")` for the root node is demanded. The evaluation traverses the AST (see Fig. 14) and collects all nodes of type `NUse` with the name `"Y"`.

The *follow* set for a nonterminal `A` is defined as the set of all terminal symbols that can immediately follow `A` in a derivation. For our CFG language as defined in Fig. 12 this can be expressed as the collection of all terminals that during derivations can follow a `NUse` named `"A"`. Fig. 16 shows the core part of the specification for the circular `follow` attribute of `NDecl` based on this definition. The value of `follow` is expressed as the union of all follow symbol sets for its use sites.

The auxiliary circular attribute `NUse.follow`, used in Fig. 16, models the set of all terminal symbols that can follow a particular use of a nonterminal. A node of type `NUse` with the name `A` corresponds to `A` appearing in a production

```
class NDecl {
   syn  HashSet occurrences(String name) =
          root().findUses(getID());
 }
 class ASTNode {
  syn  HashSet findUses(String name) {
    HashSet h = new HashSet();
    for (int i=0; i<getNumChild();i++)
      h.addAll(getChild(i).findUses(name));
    return h;
   }
}
class NUse {
  eq findUses(String name) {
    HashSet h = new HashSet();
    if (name.equals(getID()))
      h.add(this);
    h.addAll(super.findUses(name));
  }
}
```

Figure 15: Finding all uses of a nonterminal.

```
class NDecl {
  syn HashSet follow() circular [new HashSet()]{
   HashSet h = new HashSet();
   for (Iterator iter = occurrences().iterator(); iter.hasNext();) {
     NUse u = (NUse) iter.next();
     h.addAll(u.follow());
   }
   return h;
  }
}
```

Figure 16: Specification of *follow* using ordinary circular attributes.

as:

```
B ::= ... | σAγ | ...
```

Here σ and γ denote strings of terminal and nonterminal symbols. Assume that $\gamma = s_1 s_2 ... s_k$ where s_1, $s_2 \ldots s_k$ are single nonterminal or terminal symbols. The value of `follow` for this use of `A` then includes the *first* set of s_1. If s_1 is nullable, it also includes the *first* set of s_2 and so on. Should all symbols of γ be nullable, then the *follow* set of `A` also includes the *follow* set of the nonterminal of the left-hand side of the production, in this case `B`. This is modelled by the specification of `NUse.follow` in Fig. 17.

The equation for `follow` in Fig. 17 uses the auxiliary attributes `nullableSuffix` and `firstSuffix`, also declared in class `NUse`. For a particular use of a nonterminal as in the production

```
B ::= ... | σAγ | ...
```

`nullableSuffix` models the property that the empty string can be derived from the string γ. The attribute `firstSuffix` models the set of terminals that can start strings derived from γ.

The specification for circularly defined properties shown in this section does not use collection attributes. In Section 2.2 we showed how non-circular cross-referencing problems can be specified without collection attributes (see Fig. 1). It is important to stress that these solutions use different approaches. For the circular attributes, shown in this section, we have introduced an auxiliary non-circular attribute `occurrences` modelling the set of cross-references from use sites to declaration sites. Its value is used to specify the circular attribute `follow`. In this way iterations will cover only the elements of the `occurrences` set. It is also possible to specify a solution for the circular case based on the solution shown in Section 2.2. In this case, the `occurrences` attribute should be omitted. Instead, `follow` should be specified as an access to a circular attribute in the root (corresponding to the attribute `collectUses` in Fig. 1). This attribute should be specified to traverse the tree and collect the contributions from use sites. In this way iterations will cover the complete AST. In a later section we will discuss performance issues and show some experimental results. It will then become evident that the solution shown in this section for the circular attribute `follow` is much faster than a solution based on the ideas of Section 2.2.

Specification using circular collection attributes

Fig. 18 shows the declaration of `follow` in class `NDecl` as a circular collection attribute and its specification in class `NUse`. The contribution from `NUse` is expressed using the same auxiliary attribute `follow` for class `NUse` as in the previous specification using an ordinary circular attribute, i.e., the full specification

```
class NUse {
  syn HashSet follow() circular [new HashSet()];
  eq follow() {
    HashSet h = new HashSet();
    h.addAll(firstSuffix());
    if (!nullableSuffix()) return h;
    h.addAll(enclosingRule().getNDecl().follow());
    return h;
  }
  inh boolean nullableSuffix();
  inh HashSet firstSuffix();
  inh Rule enclosingRule();
}

class Rule {
  eq getProd(int index).enclosingRule() = this;
}

class Prod {
  eq getSymbol(int index).nullableSuffix() {
    for (int i=index+1; i< getNumSymbol(); i++)
      if (!getSymbol(i).nullable()) return false;
    return true;
  }
  eq getSymbol(int index).firstSuffix() {
    HashSet h = new HashSet();
    for (int i=index+1; i< getNumSymbol(); i++) {
      h.addAll(getSymbol(i).first());
      if (!getSymbol(i).nullable()) return h;
     }
     return h;
  }
}
```

Figure 17: Auxiliary attributes for specifying follow.

must also include the code of Fig. 17. However, using collection attributes, the solution no longer relies on explicitly finding all contributing sites. A specification based on collection attributes thus include the code of figures 17 and 18 while a solution based on non-collection attributes must include the code of figures 15, 16 and 17.

```
class NDecl {
  coll HashSet follow() circular [new HashSet()] with addAll;
}
class NUse
  NUse contributes
  follow()
  to NDecl.follow()
  for decl();
}
```

Figure 18: Specification of *follow* using circular collection attributes.

The specification of the *follow* set as a circular collection attribute is thus much simpler and more concise than a specification using ordinary circular attributes and demonstrates how collection attributes raise the abstraction level and the expressiveness of AGs.

7 Evaluation of Circular Collection Attributes

Evaluation algorithms that work well for circular collection attributes can be built by combining the ideas for CRAGs as in [20] and the ideas for evaluating non-circular collection attributes as described in the previous section. As a background, we will therefore briefly describe the iterative technique used in CRAGs and then show how this technique can be combined with the collection attribute algorithms presented in a previous section.

7.1 Evaluation of ordinary circular attributes

Assume that we have declared and specified an ordinary synthesized circular attribute of type `T` in a class `A` as follows:

```
class A {
  syn T s circular [init_value()];
  eq s = f(...);
}
```

In Fig. 19 an outline of the evaluator code for `s` is shown. The algorithm makes use of two global variables: `IN_CIRCLE` keeps track of if we are already inside a cyclic evaluation phase. `CHANGE` is used to check if any changes of iterative values of the attributes on the cycle have taken place during an iteration. The expressions `f(...)` correspond to the semantic function for the attribute `s`. It thus involves calls for evaluation of attributes on which `s` is dependent, some of which will be in the same cycle as `s`. The code shows the basic algorithm for circular attributes. Improvements of this algorithm are described in [20].

```
class A {
  ...
 T s_value = init_value();
 boolean s_computed = false;
 boolean s_visited = false;
 T s() {
   if (s_computed) return s_value;
   if ( ! IN_CIRCLE) {
     IN_CIRCLE = true;
     s_visited = true;
     do {
       CHANGE = false;
       Set new_s_value = f(...);
       if ( ! new_s_value.equals(s_value))
         CHANGE = true;
       s_value = new_s_value;
     } while (CHANGE);
     s_visited = false;
     s_computed = true;
     IN_CIRCLE = false;
     return s_value;
   }
   if ( ! s_visited ) {
     s_visited = true;
     T new_s_value = f(..);
     if ( ! new_s_value.equals(s_value))
       CHANGE = true;
     s_value = new_s_value;
     s_visited = false;
     return s_value;
   }
   return s_value;
 }
}
```

Figure 19: Evaluation of circular attribute `s` in class `A`.

7.2 Naive technique for Circular Collection Attributes

We first present a naive technique for evaluation of circular collection attributes. This technique builds on the ideas of the naive technique for non-circular collection attributes combined with the technique for ordinary circular attributes described above. When an instance of a circular collection attribute is demanded a tree traversal is triggered to look up and combine all its contributions. Traversal is repeated until a fixed point has been reached. An outline of the evaluator code is given in Fig. 20. Here `s` is assumed to be a circular collection attribute declared and specified as follows:

```
class A{
  coll T s() circular [init_value] with op;
}

B contributes
f(..)
to A.s()
for ref();
```

When an instance of `s` in a node `n` is demanded, it is checked whether an iterative process has already been started. This could be the case if, for example, `s` is used for defining another circular attribute, which could be either a collection attribute or a non-collection attribute. A global flag `IN_CIRCLE` is used for this purpose. If the flag is false, iterations are started and the flag is set to indicate this. In each iteration a new value of `s` is computed by traversing the tree by calling the method `nextIteration` with `n` as the parameter. The new and old values are then compared and if they are not equal, the global flag `CHANGE` is set. Iterations are continued until no changes appear. If an iterative process has already been started when `s` is demanded there are two alternatives. If it is the first time we demand `s` during an iteration (which is checked using the flag `s_visited`), a new value is computed by `nextIteration` and returned. If, instead, `s` is revisited during an iteration, its currently cached value is returned. The default behavior of the method `nextIteration` in class `ASTNode` is to perform a simple tree traversal. It is overridden in the contributing class `B`, where a contribution is combined with the current value of `s_new_value` if the `ref` attribute references the same node as the parameter.

The naive algorithm is a general method. It is always applicable when cycles occur, provided that the necessary conditions for iterations to reach a fixed point are fulfilled. It is the responsibility of the user to ensure that these conditions hold as no such checks are performed by the evaluator code.

7.3 One-phase technique for Circular Collection Attributes

It is also possible to evaluate all instances of a circular collection attribute together as in the corresponding technique for non-circular collection attributes. Using this

```
class A {
  boolean s_visited = false;
  boolean s_computed = false;
  T s_value = init_value;
  T new_s_value;

  public T s() {
    if(s_computed)
      return s_value;
    if (!IN_CIRCLE) {
      IN_CIRCLE = true;
      s_visited = true;
      do {
        CHANGE = false;
        new_s_value = init_value;
        root().A_s_nextIteration(this);
        if ( !new_s_value.equals(s_value))
          CHANGE = true;
        s_value = new_s_value;
      } while (CHANGE);
      s_visited = false;
      s_computed = true;
      IN_CIRCLE = false;
      return s_value;
    }
    if(!s_visited) {
      s_visited = true;
      T new_s_value = init_value;
      root().A_s_nextIteration(this);
      if ( !new_s_value.equals(s_value))
        CHANGE = true;
      s_value = new_s_value;
      s_visited = false;
      return s_value;
    }
    return s_value;
  }
}
class ASTNode {
  void A_s_nextIteration(A n){
    for(int i = 0; i < getNumChild(); i++)
      getChild(i).A_s_nextIteration(n);
  }
}
class B {
  void A_s_nextIteration(A n) {
    if(ref() == n)
      n.new_s_value.op(f(...));
    super.A_s_nextIteration(n);
  }
}
```

Figure 20: Naive evaluation of circular collection attribute `s` in class `A`.

technique, tree traversals will be repeated until there are no changes of the values of any instance. We have, however, not implemented this algorithm in JastAdd. One reason is that it may enforce evaluation of instances belonging to different strongly connected components in the dependency graph. As was shown in [20] for circular attributes it is desirable to iterate over different components in separate iteration processes. Such components most probably require different number of iterations to reach a fixed point. If two or more are iterated together, it will be necessary to perform iterations for all of them until the component requiring the maximum number of iterations has converged. Efficiency will thus deteriorate. For ordinary circular attributes we have therefore introduced an optimization technique which in some cases can detect different strongly connected components during iterations. When a new component is entered, iterations of the previous one is suspended. When the second component has been iterated, fixed point iteration in the first component is resumed.

Assume that the one-phase technique is implemented by simply repeating tree traversals until the fixed points for all instances involved have been reached. Then, during traversal ordinary circular attributes may be demanded. The evaluator code for such attributes involves code to detect separate components and to suspend iterations temporarily. It also includes code to detect if components that have been suspended are incorrectly re-entered. This can happen if the AG author has forgotten to declare a circularly defined attribute as circular, and it may cause incorrect values to be returned if it is not detected. In the one-phase technique, however, we must allow certain separate components to be evaluated together, namely those to which the instances of the collection attribute under computation belong. Further investigations are needed to find out how a robust one-phase technique, detecting errors of the type mentioned, can be designed in combination with the optimization technique for non-collection attributes.

We have tested the algorithm for cases where no interactions with ordinary circular attributes take place. Results indicate that the efficiency of the one-phase technique is inferior to the two-phase technique which will be described below. This, of course, makes it less interesting to make an effort to design a robust one-phase algorithm.

7.4 Two-phase technique for Circular Collection Attributes

This algorithm builds on the ideas of the two-phase joint evaluation technique used for non-circular collection attributes. The survey phase is carried out as before while the combination phase is iterated until a fixed point is reached. As for non-circular attributes the algorithm has two variants. One evaluates conditions during the survey phase and the other postpones these computations until the combination phase. In Fig. 21 the evaluator code for the declaring class `A` is shown. Evaluator code for the root class `Program`, the base class `ASTNode` and the declaring class

`B` is identical with the code for these classes for the two-phase algorithm for non-circular collection attributes as shown previously in Fig. 7.

Applicability of the two-phase circular algorithms

For both variants of the two-phase technique, the survey phase is carried out in a non-iterative manner. As described before, in connection with the two-phase technique for non-circular collection attributes, dependencies between instances of the collection attribute under computation appearing during this phase will cause a looping behavior. For the 2Ph-LC variant this occurs when any of the reference attributes used for giving contributions is dependent on an instance of the collection attribute. In these cases one has to resort to the naive algorithm. Thus, the 2Ph-LC algorithm for circular collection attributes is not equivalent to the naive technique with respect to applicability as was the case for their non-circular counterparts. For the 2Ph-EC variant, looping behavior also occurs if any of the conditions introduces this type of dependency.

It is worth mentioning that, for un-grouped evaluation techniques, we have not found any practical cases where circularities arise due to the reference attributes. For practical use, we therefore think it is not likely that the faster 2Ph-LC needs to be replaced by the general naive algorithm.

The evaluator code can be modified to detect and report failure during the survey phase in the same manner as for the two-phase technique for non-circular collection attributes, shown earlier.

8 Performance of Collection Attribute Algorithms

In this section, we first compare efficiency for solutions using ordinary attributes with those using collection attributes. Then, in the following subsections the different evaluations schemes for collection attributes are compared for efficiency.

8.1 Comparing solutions using ordinary attributes with those using collection attributes

In Section 2 we showed how an attribute, collecting the set of use sites for local variables, could be specified using ordinary attributes. After describing evaluation techniques in previous sections it should now be obviuos that this solution emulates the naive technique for evaluation of collection attributes. We will therefore refer to the specification using ordinary attributes as an *emulated collection attribute* solution.

As an extension to the front end of our JastAdd extensible Java compiler [9], we have specified two cross-reference attributes: `varUses` which is similar to the computation of `uses` in Section 3.2, but uses a condition in the contribution to cover uses of variables only; and `subClasses` as shown in Section 5.1. To

```
class A {
  boolean s_visited = false;
  boolean s_computed = false;
  T s_value = init_value;
  HashSet s_contributors = new HashSet();

  public T s() {
    if(s_computed)
      return s_value;
    root().collect_contributors_A_s();
    if (!IN_CIRCLE) {
      IN_CIRCLE = true;
      s_visited = true;
      do {
        CHANGE = false;
        T new_s_value = init_value;
        combine_s_contributions(new_s_value);
        if ( !new_s_value.equals(s_value))
          CHANGE = true;
        s_value = new_s_value;
       } while (CHANGE);
       s_visited = false;
       IN_CIRCLE = false;
       return s_value;
    }
    if(!s_visited) {
      s_visited = true;
      T new_s_value = init_value;
      combine_s_contributions(new_s_value);
      if ( !new_s_value.equals(s_value))
        CHANGE = true;
      s_value = new_s_value;
      s_visited = false;
      return s_value;
    }
    return s_value;
  }

  private T combine_s_contributions(T c) {
    for(Iterator iter = s_contributors.iterator(); iter.hasNext(); ) {
      ASTNode contributor = (ASTNode) iter.next();
      contributor.contributeTo_A_s(c);
    }
    return c;
  }
}
```

Figure 21: Two-phase evaluation of circular collection attribute `s` in class `A`.

measure performance, we have run a number of Java programs in this extended front end. Table 1 shows the results for demanding all instances of `varUses`, and Table 2, the same for `subClasses`.

The tests have been run on three sample Java programs. The first one is a typical student program (stud) with about 750 lines of code. The second program is an artificial test program (art test) of 15k lines specially constructed to test a case with several contributions for each collection attribute instance. The third program is the source code for the javac compiler which comprises about 36k lines.

Time is measured in milliseconds and is given for the following different evaluation alternatives: Emulated collection attributes (Emul attrs), naive algorithm (Naive alg), two-phase joint evaluation with late evaluation of conditions (2Ph-LC) and one-phase joint evaluation (1Ph).

For each test, the following information concerning the size is given: the number of lines of Java code in the program (lines), the number of collection attribute instances (coll inst), the total number of potential contributors for all instances of the collection attributes, i.e., the number of instances of the contributing node class (contr inst) and the number of valid contributions (valid contr), i.e., the number of contributions for which the attached condition, if any, is true. Note that the number of valid contributions can be greater than the number of contributing nodes when there are for-each clauses or when-clauses in the contributions.

Improvement factors (Impr factor) are given for the emulated algorithm versus the naive algorithm and for the naive algorithm versus the 2Ph-LC algorithm.

Table 1: Computation of varUses

	Size				Time (ms)				Impr factor	
Program	**lines**	**coll inst**	**contr inst**	**appl contr**	**Emul attrs**	**Naive alg**	**2Ph-LC**	**1Ph**	**Emul/ Naive**	**Naive/ 2Ph-LC**
stud	750	73	337	184	4100	1050	50	23	3.9	21
art test	15000	30	14565	14565	3100	730	135	55	4.2	5.4
javac	36500	1969	29180	6464	675000	165000	210	215	4.1	786

Table 2: Computation of subClasses

	Size				Time (ms)				Impr factor	
Program	**lines**	**coll inst**	**contr inst**	**appl contr**	**Emul attrs**	**Naive alg**	**2Ph-LC**	**1 Ph**	**Emul/ Naive**	**Naive/ 2Ph-LC**
stud	750	8	8	0	550	160	35	30	3.4	4.6
art test	15000	77	77	126	6300	1600	60	50	3.9	27
javac	36500	180	180	153	110000	15000	200	220	7.3	75

As is evident from the execution times given in the tables, the naive technique substantially improves the performance as compared to the emulated solu-

tion, speeding up evaluation by a factor of 3-7 for the given tests. The main reason being that the synthesized and inherited attributes in the latter specification must use declarative specifications of partial results, whereas the generated code for the naive algorithm can represent partial results as variables that are updated during the traversal. Also, the naive technique for collection attributes is outperformed by the alternative techniques. This will be further discussed and motivated in the next section.

8.2 Comparing techniques for non-circular collection attributes

It will, in general, not be possible to state that a certain algorithm for evaluating collection attributes is always faster than another algorithm. Some applications may, for example, not require all instances of an attribute to be evaluated while others do. In the former case an algorithm using a pure demand-driven technique might be the fastest while in the latter case an algorithm deviating from pure demand evaluation might be more efficient. When discussing individual algorithms and variants of algorithms below, we will therefore limit the description to results obtained for practical applications. In some cases we will also briefly discuss performance for extreme cases, for example when only a single instance of a collection attribute is needed .

Naive versus one-phase techniques. In the naive technique, a complete tree traversal is performed for each demanded instance of a collection attribute. It is a pure demand-driven technique as nothing is computed until it is needed. The one-phase technique, on the other hand, performs only one tree traversal during which all instances are computed and cached. These computations are carried out when, for the first time, an instance of the collection attribute is needed. The one-phase technique is thus not purely demand driven. As a consequence, difference in efficiency between the two techniques depends on how many instances of the attribute are actually needed.

In many applications only o few instances are needed. But since traversal cost dominates, the one-phase technique will be faster as soon as a few instances are demanded. The experiments shown in Tables 1 and 2, where all instances are demanded, verify this.

For an extreme case, when only one instance is needed, the naive technique is the fastest. Both algorithms perform one tree traversal in this case. The one-phase technique, however, performs many unnecessary computations during this traversal.

Two-phase techniques versus the naive technique. The two-phase technique is not purely demand driven. When, for the first time, an instance is demanded it also prepares for subsequent computations of other instances in its first phase.

If all n instances are demanded, the naive technique performs n tree traversals while the two-phase techniques traverses the tree only once. If n is sufficiently large, the two-phase techniques therefore outperforms the naive technique, which is verified by the results in Tables 1 and 2.

Clearly, if only one instance is needed, then the two-phase algorithm, (especially its 2Ph-EC variant) performs unnecessary computations during the first phase and will therefore be slower than the naive technique.

2Ph-LC versus 2Ph-EC. First, consider each of the two phases separately:

- When there are conditions, it is reasonable to expect the survey phase of the 2Ph-EC variant to be slower than that of 2Ph-LC. If there are no conditions the survey phase of both variants are equivalent. Thus, the survey phase of the 2Ph-EC is never faster than that of the 2Ph-LC variant.
- During the second phase, the value of an instance is computed by iterating over the contributor set. If the 2Ph-EC variant is used the contributor sets will always be smaller than or equal to those of the 2Ph-LC variant. The second phase of the 2Ph-EC technique will therefore always be faster than or equally fast as the corresponding phase of the 2Ph-LC technique.

To summarize, it is not possible to generally state which of the two variants is the fastest. For cases where the first phase of the 2Ph-EC variant is considerably slower than that of the 2Ph-LC while their second phases are equally fast, the 2Ph-LC variant is the most efficient. On the other hand, for cases where conditions are simple to evaluate and are extremely filtering, the 2Ph-EC technique can be expected to be the fastest since its second phase will iterate over much smaller sets and thereby perform significantly fewer method calls.

We have measured the two variants, and found the differences in execution times to be very marginal.

Two-phase versus one-phase technique. If all instances are demanded, then both variants perform one tree traversal and compute all conditions and all valid contributions. In addition, the two-phase algorithms compute auxiliary sets of contributors and iterate over these. It might therefore be expected that the one-phase algorithm is somewhat faster in this case. However, for large ASTs the tree traversal dominates the execution time and the additional work carried out by the two-phase techniques becomes insignificant. Actual execution times can then be due to details in the implementation of the algorithms. For the examples shown in Tables 1 and 2, where all instances are

demanded, the one-phase algorithm is the fastest for the two smallest programs while the two-phase technique seems to be a little faster for the largest of the programs.

If only one of several instances is needed, then clearly the one-phase joint technique performs many unnecessary computations as all attribute instances are fully evaluated.

The two-phase technique will remain the most efficient as long as sufficiently few instances of a collection attribute are demanded. A reason why instances can be un-demanded is that for some analysis tasks only instances appearing in certain contexts are of interest. This is the case, for example, in devirtualization analysis.

Table 3 shows results for a very simple devirtualization analysis. It checks the two simple conditions for possible devirtualization described in Section 5.1 (Zero subclasses and No overriding methods). The specification involves two collection attributes modelling the set of subclasses for a class and the set of overriders of a method. As indicated in the table, only a subset of these attribute instances are demanded. If a method is never called or if it only appears in static method calls, the value of its collection attribute instance is not needed. As a consequence, the 2Ph-LC is somewhat faster in this case.

Table 3: Devirtualization computations

Size				**Time (ms)**	
Coll attr instances		**Demanded instances**		**2Ph**	**1Ph**
subClasses	**overriders**	**subClasses**	**overriders**	**LC**	
645	6769	139	1219	700	760

Grouped versus ungrouped techniques. We have implemented grouped variants for both the two-phase and the one-phase joint technique.

When there are several attribute instances and all or a majority are demanded, the grouped variants are faster. The reason being that only one single tree traversal will be performed during evaluation while the ungrouped variants perform one traversal for each member of the group.

We have performed measurements on our implementation of the Chidamber and Khemerer metrics, described in Section 5.2. Since these are defined using seven different collection attributes, this gives the opportunity to measure grouped evaluation, i.e., evaluating several different collection attributes jointly.

As a sample application to compute the metrics on, we have used the source code of the Jigsaw web server. Jigsaw is the W3C's web server platform, consisting of more than 100k lines of Java code excluding comments.

We have performed one suite of tests computing metrics for *all* types in the Jigsaw application, and another suite of tests for only the types that are in packages starting with *org.w3c.www*. While this subset of types accounts for roughly a fifth of the source code, all source code for Jigsaw is needed to perform the computations, since some of the metrics take contributions from types in other Jigsaw packages.

This allows us to evaluate whether we can exploit the demand driven evaluation of the collection attributes for this particular application: For the first suite, all collection attribute instances are demanded. For the *org.w3c.www* case, only a subset of the instances are demanded.

Table 4 shows the results of our experiments for different evaluation algorithms. We see that the one-phase algorithm (1Ph) is somewhat faster than the two-phase algorithm (2Ph-LC), even in the *org.w3c.www* case when not all collection attribute instances are demanded. The number of demanded instances is thus not small enough for the two-phase algorithm to get an advantage. The contributions are actually all very simple, typically adding one to a counter or adding a reference to a set. The main execution cost is thus related to AST traversal.

The times in Table 4 are in milliseconds and improvement factor (*Impr factor*) is the quota between the execution times for ungrouped and grouped evaluation. Times include only the execution time for computing the metrics. The entire analysis also includes lexing, parsing, AST building, and error checking, which adds another 12 seconds to the overall analysis time. It is somewhat unfair to compare this result to *ckjm* [22], which takes 3.4 seconds, since it processes bytecode which requires much less static analysis, e.g., all names are bound beforehand. Processing source will therefore always be more expensive than bytecode, but we still find the performance perfectly reasonable for a fairly large project, and we notice that for this particular application the bottleneck is not the collection attributes.

Table 4: Metrics computation

	Time (ms)		**Impr factor**	**Time (ms)**		**Impr factor**
Jigsaw types	**1Ph**	**1Ph-Grp**	**1Ph/ 1Ph-Grp**	**2Ph-LC**	**2Ph-Grp**	**2Ph/ 2Ph-LC-Grp**
all	2207	1535	1.4	2585	1948	1.3
org.w3c.www	1320	693	1.9	1639	1002	1.6

8.3 Comparing techniques for circular collection attributes

We have implemented two algorithms for evaluation of circular collection attributes, the naive technique and the two-phase technique. As for the non-circular case the latter has two variants, one where conditions are evaluated during the first phase and one where these computations are postponed until the second phase.

Naive versus two-phase techniques. It is reasonable to assume that the iterative process dominates the execution time of the algorithms. In most cases, the number of contributing nodes are significantly smaller than the number of nodes of the AST. Iterations will then cover smaller sets in the two-phase algorithms than in the naive algorithm. The two-phase algorithm can therefore be expected to be the fastest.

2Ph-LC versus 2Ph-EC. Again we will assume that the iterative process dominates execution times. In the EC algorithm, the contributor sets are always smaller than or equal to the corresponding sets in the LC variant. Iterations will cover the elements in these sets. Therefore the LC variant will be less efficient or equally efficient as the EC variant. The latter case occurs when contributor sets are of equal size for both methods and this will be the case when there are no conditions attached to contributions or when all such conditions evaluate to true.

We have compared different solutions for computation of the *follow* set for context-free grammars (see Section 6). Table 5 shows execution times for the 2Ph-LC and the naive techniques used to compute the *follow* sets for the Java grammar.

For comparison, we also show results for three solutions using ordinary circular synthesized attributes.

Emul-naive specifies the `follow` attribute through a tree traversal during which contributions to *follow* sets are collected. It is based on the emulated technique used in Section 2.2. The evaluator will in this case perform iterations over the entire AST.

Emul-improved1 is the solution shown in Section 6. It uses an auxiliary non-circular attribute `occurrences` to model the cross-reference sets containing references from nonterminal use sites to their corresponding declaration sites. The circular attribute is specified as the union of the follow sets for all elements in `occurrences`. As a consequence, iterations will only include contributing nodes.

Emul-improved2 is the one used in [20]. It uses the same technique as Emul-improved1, but tree traversal is expressed in an imperative manner as outlined in Fig. 22. Here `collectUses(HashSet h, String name)` is an ordinary Java method declared in class `ASTNode`. Its default behavior,

performing a tree traversal, is overridden in class `NUse` where a use instance adds itself to `h` if its name matches the parameter `name`. By using imperative code in this way (which is allowed in JastAdd) there is no longer a need to expose the partial collections of use sites as attributes and to represent them as separate objects. Letting equations use imperative code might, in general, be unsafe. However, in this case (Fig. 22) the usage of imperative code is safe since the code affects only local data inside the equation, i.e., it has no externally visible side-effects.

```
class NDecl {
   syn  HashSet occurrences(String name) =
      root().findUses(name);
 }
class CFG {
  syn  HashSet findUses(String name) {
    HashSet h = new HashSet();
    collectUses(h,name);
    return h;
  }
}
class ASTNode {
  protected void  collectUses(HashSet h, String name) {
    for (int i=0; i<getNumChild();i++)
      getChild(i).collectUses(h,name);
   }
}
class NUse {
  protected void collectUses(HashSet h, String name) {
    if (name.equals(getID()))
      h.add(this);
    super.collectUses(h,name);
  }
}
```

Figure 22: Finding all uses of a nonterminal using an ordinary Java method.

The Emul-naive and the Emul-improved1 solutions are outperformed by solutions using collection attributes. When comparing the Emul-improved1 with the naive technique for collection attribute this might be surprising, since the latter performs iterations the entire AST while the former iterates over smaller sets. The explanation is that the declarative approach in Emul-improved1 makes it necessary to represent partial sets as separate objects. The generated evaluator code for the naive technique, on the other hand, uses ordinary Java methods to update sets during tree traversals.

The Emul-improved2 specification is, however, faster than the naive algorithm for collection attributes. The reason is that it does not need to represent partial sets by separate objects. This, together with the fact that it iterates over smaller

sets than the naive technique for collection attributes, explains the result. Also, Emul-improved2 solution is only marginally slower than the 2Ph-LC algorithm. The experiment thus indicates that circular collection attributes give modest performance improvement in comparison with solutions based on non-collection attributes when parts of their specification are realized by ordinary methods. Their main advantage in these cases, is that solutions using collection attributes yield much simpler specifications.

Solutions involving circular set-valued attributes are particularly sensitive to choice of data structure. The reason is that many set operations are performed during iterations. In Table 5 we show execution times for two alternative set representations: `HashSet` from the java.util class library and the more efficient `BitSet` which is based on bit vectors. As is evident from the table, a more efficient data structure results in substantially shorter execution times for all alternative solutions.

Table 5: Circular collection attributes

	Size		**Time (ms)**				
Data structure	**Coll inst**	**Contr inst**	**Emul-naive**	**Emul-improved1**	**Emul-improved2**	**Naive**	**2Ph-LC**
HashSet	155	280	22500	4900	320	480	270
BitSet	155	280	13500	2200	170	270	70

8.4 Choice of algorithm

The algorithms presented for non-circular and circular collection attributes differs with respect to applicability as was pointed out in Section 4.6 and 6 and with respect to performance as described earlier in this section.

JastAdd allows the user to choose algorithm for individual attributes or groups of attributes by annotating their declarations. If no annotation is given, the 2Ph-LC algorithm is used for non-circularly declared attributes and its corresponding fixed-point iterating variant for attributes declared as circular.

Choice of algorithm must first and foremost be based on applicability. If one algorithm works well for a certain application, then another less general algorithm might be tried in order to speed up computations. For example, if no dependencies exist between instances of a collection attribute and if all instances are expected to be demanded during computations, then the 1Ph-Joint is a good choice. The grouped variant might speed up computations further. Should there be any inter-dependencies between instances in the group, the evaluator will report this.

9 Related Work

Restricted forms of collection attributes were introduced by Knuth [18] who allowed global sets in the start symbol, and by Kaiser [15] and Beshers [3] who allowed collection attributes associated with subtrees. Hedin introduced general collection attributes with contributions via reference attributes, but with partly manual implementation techniques [12].

The collection attributes as presented in this paper were introduced by Boyland in his PhD-thesis, [4]. His APS system also supports circular collection attributes.

The focus of Boyland's thesis is on the attribute specification mechanisms and how they can be applied. There is only a brief sketch of the implementation, and no performance results are reported. Just like JastAdd, the APS system uses a demand-driven evaluation technique. The APS technique for evaluating collection attributes is based on a concept called *guards*, i.e., artificial attributes that are added by the APS compiler. Consider a collection attribute c in node type N. Each instance of c is made dependent on a guard, and the guard is in turn made dependent on all the reference expressions of type N. Each instance of c is evaluated on demand, but not until all reference expressions of type N are evaluated. The implementation sketch does not give the details of how this is done, but the effect seems similar to our two-phase algorithm.

In his later work on collection attributes, e.g., [5], Boyland investigates static evaluation algorithms and incremental versions of them. In this work, circular dependencies are no longer supported, and the evaluation technique is based on static analysis of the grammar rather than demand evaluation. This work focuses on theory and contains no reports on practical applications or performance results.

Silver [24] is a recent AG system supporting collection attributes. It supports several extensions to traditional AGs such as higher-order attributes, forwarding and pattern matching. There is, however, no support for circular attributes. In Silver, attributes are evaluated by translating the AG specifications into Haskell. The system is modular, consisting of a core attribute grammar language which serves as the host language for specifying extensions. Collection attributes have been implemented as one such language extension. The work focuses on the composability of language constructs and the application of Silver to extensible and domain-specific languages. No performance results or specific evaluation algorithms are reported.

10 Conclusions

We have shown how cross-reference-like properties can be specified very concisely using collection attributes. We have presented several evaluation algorithms, and found that the joint evaluation and two-phase algorithms work very well for large practical applications. Our implementation of the Chidamber and Kemerer metrics increased the compilation time with only 1-2 seconds for Java programs of 100 k lines of code. As shown by our smaller examples, the naive algorithm is several

orders of magnitude slower. Emulation, using ordinary inherited and synthesized attributes, is much slower still.

We have presented a series of variants on joint evaluation algorithms. The 2Ph-LC variant is a general algorithm and can handle all non-circular collection attributes. It is used by default in the JastAdd system. The other variants: 1Ph and 2Ph-EC, can be faster for some applications, but they are not completely general: contrived non-circular examples can be constructed that these algorithms cannot handle. For our example applications, the 2Ph-EC variant was only very marginally faster than the 2Ph-LC algorithm. The 1Ph variant was marginally slower on some examples and marginally faster on others. These variants can be selected by annotating individual collection declarations.

Using grouped joint evaluation, performance improved by a factor of between 1.3 and 1.9 on the metrics application, depending on source program and on algorithm variant. However, grouping cannot be done automatically: the user has to explicitly annotate the collection attribute declarations with the desired group.

We have also extended the algorithms to circular variants that can handle collection attributes explicitly declared as circular. The 2Ph-LC circular algorithm is used as the default. On our example application in grammar flow analysis, the use of circular collection attributes gave a much simpler specification than the corresponding non-collection attribute solution, without leading to decreased performance.

Bibliography

[1] JastAdd, 2007. Available at: http://www.jastadd.org.

[2] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *OOPSLA*, pages 324–341, 1996.

[3] G. M. Beshers and R. H. Campbell. Maintained and constructor attributes. In *Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments*, pages 34–42. ACM Press, 1985.

[4] J. T. Boyland. *Descriptional Composition of Compiler Components*. PhD thesis, University of California, Berkeley, 1996.

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

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[7] David Detlefs and Ole Agesen. Inlining of virtual methods. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 258–278. Springer-Verlag, 1999.

[8] T. Ekman. *Extensible Compiler Construction*. PhD thesis, Lund University, Sweden, 2006.

[9] T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. Accepted for publication at OOPSLA'07.

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

[11] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of CC'86*, pages 85–98. ACM Press, 1986.

[12] G. Hedin. An Overview of Door Attribute Grammars. In *Proceedings of CC'94*, volume 786 of *LNCS*, pages 31–51, Edinburgh, 1994.

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

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

[15] G. E. Kaiser. *Semantics for structure editing environments*. PhD thesis, Carnegie Mellon University, 1985.

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

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP 2001*, volume 2072 of *LNCS*, pages 327–355. Springer, 2001.

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

[19] E. Magnusson, T. Ekman, and G. Hedin. Extending Attribute Grammars with Collection Attributes - Evaluation and Applications. 2007. Accepted for publication at SCAM 2007.

[20] E. Magnusson and G. Hedin. Circular Reference Attributed Grammars - Their Evaluation and Applications. *Science of Computer Programming*, 68(1):21–37, 2007.

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

[22] Diomidis D. Spinellis. ckjm - Chidamber and Kemerer Java Metrics, 2007. http://www.spinellis.gr/sw/ckjm/.

[23] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. *SIGPLAN Not.*, 35(10):264–280, 2000.

[24] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an Extensible Attribute Grammar System. In *Proceedings of the 7th Workshop on Language Descriptions, Tools, and Applications (LDTA'07) at ETAPS 2007*, 2007.

[25] E. Van Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute Grammar-based Language Extensions for Java. In *Proceedings of ECOOP'07*, LNCS. Springer, 2007.

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

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