Applications and Extensions of Reference Attributed Grammars

Eva Magnusson

Licentiate Thesis, 2003

Department of Computer Science
Lund Institute of Technology
Lund University

ISSN 1404-1219
Dissertation 19, 2003
LU-CS-LIC:2003-3

Thesis submitted for partial fulfillment of
the degree of licentiate.

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

E-mail: Eva.Magnusson@cs.lth.se
WWW: http://www.cs.lth.se/~eva

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 tools that manipulate programs, e.g., compilers, smart language-sensitive editors, software metric tools, etc. Our 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 evaluators.

The thesis consists of an introduction and three 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 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 techniques have all been implemented and tested in practice.

Preface

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

The research papers included in 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. LDTA'03, Third Workshop on Language Descriptions, Tools and Applications, Warsaw, Poland. April 2003. To appear in ENTCS, Electronic Notes of Theoretical Computer Science, Elsevier.

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.

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. Much 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.

The 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. Torbjörn Ekman and Anders Nilsson have both been active in ongoing projects involving the JastAdd tool and discussing with them has helped me in various ways. Christian Andersson and Christian Söderberg both helped me to prepare final versions of papers for publication by sharing their expertise in Latex. Ulf Asklund took time to help me a lot with Framemaker when preparing this thesis. Per Holm has also patiently shared his experience in using Framemaker. 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 who encouraged and helped me in so many different ways. There is a proverb that says: “Gratitude takes three forms: a feeling in the heart, an expression in words, and a giving in return.” I have come to the conclusion that if I tried the second form, putting it into words, it would almost certainly only result in a masterpiece of understatement. So, I will keep myself to the first part and do my best to comply with the third part.

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

Contents

Paper III: Circular Reference Attributed Grammars – their Evaluation and Applications 65

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 tools that manipulate programs, e.g., compilers, smart language-sensitive editors, software metric tools, etc., where the basic data structure is an abstract syntax tree.

Our basis is Reference Attributed Grammars (RAGs) [7], 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) [16] 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 some aspects of compiler related tasks, for example name- and type-analysis of languages with complex scope rules. By allowing references between distant nodes in the AST, the RAG formalism facilitates these tasks. The object-oriented view of the grammar used in RAGs is a conceptual extension of AGs which make it possible to apply all the object-oriented 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 are formal, e.g., allowing cyclic dependencies between attribute instances, 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 to express while their equivalent imperative style implementations are almost trivial. 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 aspect-oriented modularization and where modules can be implemented in either imperative or declarative style.

The thesis consist of an introduction and three 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 open up the possibility to combine imperative and declarative style modules.
- *Extensions of RAGs*, how they widen their applicability 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 aspect-oriented programming and the combination of imperative and declarative programming. The third paper describes a formal extension of RAGs supporting circular dependencies (CRAGs). The extended formalism opens up new application areas which is also discussed in the paper.

The techniques have all been implemented and tested in practice. Our tool JastAdd incorporates the extended AG formalisms. Its modularization concept also allows imperative style modules to be freely combined with AG modules. An automatic evaluator construction capability that can handle the extended formalisms has also been developed.

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

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 we discuss the objectives of our work and present the papers. Finally, in Section 4, we conclude and discuss some possible future work.

1 Attribute Grammars

1.1 Traditional Attribute Grammars

Traditional attribute grammars (AGs) were introduced by Knuth [16] 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. Declarative specifications are clear and concise and help avoiding errors common in imperative style programming.

Consider a production $X_0 ::= X_1\ X_2 ... X_k$ 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*. Here a_0 must be either a synthesized attribute of X_0 or an inherited attribute of X_j, $1 \le j \le k$. The arguments of the semantic function, $a_1, ..., a_n$, are attributes of X_j, $0 \le j \le k$. Each equation thus only involves information associated with the symbols of the production. An equation defines either a synthesized attribute of the nonterminal of the left-hand side or an inherited attribute of one of the right-hand side nonterminals.

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 nonterminals X_j, $1 \le j \le k$. 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 of a_1. Traditional AGs consider cyclic dependencies as an error, i.e., for any syntax tree derivable from the grammar there must not be any cyclic 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., [2,7,20], 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.

There has also been suggestions of allowing remote attribution, i.e., to define attributes in distant nodes. Boyland [2] has a system allowing remote access and remote definition for collection-valued attributes like sets and dictionaries.

In our work we have used Reference Attributed Grammars (RAGs) as suggested by Hedin [7]. This extension supports attribute access via references but not remote attribution. 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 [7], design pattern checking [3], and prediction of worst case execution times [19].

RAGs were initially implemented in a tool APPLAB [1], developed at our department. APPLAB is an interactive environment based on language-sensitive editing, aimed at the interactive design of domain-specific languages. We used APPLAB for the implementation of the work described in the first paper of this thesis. In our second paper we describe

our new tool JastAdd, which is an aspect-oriented compiler construction tool supporting RAGs, aspect-oriented modularization, as well as the combination of imperative style and declarative style modules. The JastAdd tool has also later been enhanced to handle circular grammars as described in the third paper.

1.3 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, e.g., see [5,10].

In a traditional AG it is considered an error if cyclic 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 [5,10].

The most common way to ensure that circular AGs are well defined is to require that the domain of attributes involved in cyclic definitions are 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 Figure 1.

```
initialize all attributes x involved in the cycle to a bottom value;
do {
    for each 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. 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 [5] 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 non-circular specification, in contrast, adds huge complexity using, e.g., higher order functions one of which in essence captures the iterative process used in Figure 1.

In the third paper of this thesis we address the possibility and advantages of combining circular attribute grammars with reference attributed grammars.

1.4 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 [6]. 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 not corresponding to any of the nonterminals of the grammar. For example, it might be convenient to introduce an abstract class Any 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 Any. Figure 2 shows how these object-oriented concepts are used to equip every node in an abstract syntax tree with a reference attribute env referencing the Block node corresponding to the closest enclosing block. Equations of type

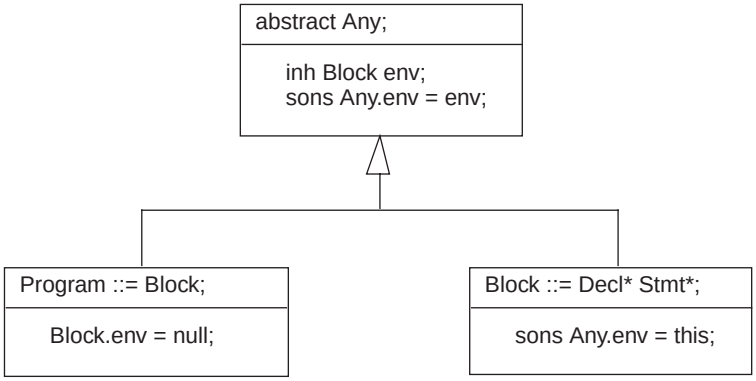


Figure 2 *Specifying a* env *attribute for all nodes in an AST*

sons Any.env = ...; are so called *collective equations*, expressing that the definition applies to all son nodes of type Any. The default behavior is that child nodes inherit their parent's environment as stated by the equation for the nonterminal Any. This definition is then overridden for the start nonterminal Program whose child of type Block has no environment. It is also overridden in the equation of Block, whose children have the block node (this) as their environment.

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 stores attribute instances in memory cells and evaluates them in an order corresponding to the topological sort of the dependency graph. In a *demand-driven* technique each attribute is replaced by its semantic function. Accessing an attribute is here realized by calling its semantic functions.

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

2.1.1 Static techniques

Static techniques 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 [14] and Courceller & Franchi-Zannettacci [4]. In their scheme all possible dependencies between attributes of each production are derived. 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, e.g., Ordered Attribute Grammars [13].

2.1.2 Dynamic techniques

Dynamic techniques analyze dependencies between attribute instances at evaluation time for the abstract syntax tree at hand.

An example of a general dynamic technique is the one proposed by Jones [10]. 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.

2.1.3 Demand-driven techniques

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 [8,11,17]. The drawback is that it 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 but the amount of memory space needed might be substantial. An alternative is to let the AG author decide which attributes to cache.

The object-oriented view on attribute grammars presented in Section 1.4 makes it very easy to automatically generate demand-driven evaluators. In fact, the evaluator is implicitly constructed directly by the modelling of the attributes and their equations as 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 2.1.3 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 [1] and by our new JastAdd tool described in paper 2, as well as by other systems for similar formalisms, e.g., [2,20].

2.3 Evaluation of Circular Attribute Grammars

Farrow [5] showed that the static technique of Katayama can easily 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 of each other and they are evaluated together by an iterative process as described in Figure 1.

Jones [10] showed that his dynamic technique can also easily 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 gram-

mars it thereby becomes a technique that is capable of evaluating grammars of the combined extended formalism, circular reference attributed grammars (CRAGs).

3 Objectives of this Work and Description of the Papers

The research presented in the papers of this thesis have the following objectives:

- *Applications*
  The traditional application area for AGs is related to compiler construction. Our aim 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 and 3.

  The first paper focuses on a new application for RAGs; program visualization. In the third paper we show how extending the formalism to allow circular dependencies 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 almost trivial. 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. They 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.

- *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.

  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.

The rest of this section briefly introduces the three included papers.

3.1 Paper 1: Program Visualization using Reference Attributed Grammars

Traditional applications for attribute grammars are 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 [1] that has been developed earlier at our department. The tool supports interactive development of application-specific languages and 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. For example, for a state transition language the node classes could be State and Transition. Attributes for these classes are introduced and specified to model the properties 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. 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 paper exemplifies the technique by using a state transition language as a running example. It is first shown how the visualization front-end, backend and glue module for a very simple toy language, TinyState, can be specified. Then it is shown how a visualization of a more complicated state transition language, ExSpecState, can be realized by writing a new glue module only.

The scope of this application is restricted to static code visualization. It is language independent in the sense that it can be reused for different languages by specifying different glue modules. It is also visualization tool independent in the sense that if a new tool is to be used only one module 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. RAGs are also used implicitly by the glue module which uses the static semantics module of the language at hand. This will typically include a name analysis module which uses reference attributes to tie use sites to declaration sites.

The work on which this paper is based was done before the implementation of the tool JastAdd, described in the next paper. Some of the specifications for the visualization were difficult to express in the declarative paradigm while their corresponding imperative implementations would have been almost trivial. 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.

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 [9]. 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 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 1.4) 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 JastAdd version presented in this paper supports reference attributed grammars. It has later been generalized to support circular reference attributed grammars. This is described in the next paper.

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, lifts 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 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 [18,12]. 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 in a number of practical problems. We have also compared our demand-driven evaluator with handwritten imperative code implementing fixed-point iterations (in JastAdd) and found that there seems to be little difference in performance.

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, 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 formalism 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.

Paper 1

We, as well as other researchers [3,19], 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 facili-

tates 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. A CRAG is a combination of two formal 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 have demonstrated how these solutions can be almost directly formulated as CRAG specifications.

Furthermore, CRAGs widens 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.

4.2 Future work

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

Modularization

The modularization concept used in our work has so far only been a computational modularization, i.e., a way to textually separate computations. An interesting field for further research is extending the modularization concept to include support for language extensions and *language modularization*.

Aspect-oriented programming, e.g., as in AspectJ [15], covers both static and dynamic aspects. Our tool JastAdd presently only covers static aspect-oriented programming by allowing fields, method and interface implementations to be added in separate modules. The *joinpoint* model of AspectJ allows also code written in separate modules to be inserted dynamically at selected execution points. It is a very interesting area for future research to investigate further the benefits of *adding dynamic aspects*, especially for compiler construction.

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 achieved by caching all attributes. It would be desirable to develop a technique for automatically deciding which attributes to cache for best performance and memory usage.

The technique used in our tool JastAdd is based on a straight-forward translation of attributes and their equations into methods and fields in Java and the resulting evaluator perform many method calls. Possible *optimizations of the evaluator implementation* include declaring methods `final` when possible and to explore the possibility of inlining methods.

Another possibility would be to apply some of the faster *static evaluation techniques* based on dependency graphs described in Section 2 instead of demand-driven evaluation. That would, however, require an initial dependency analysis. It is an open question to what extent it is possible to perform a realistic dependency analysis for RAGs.

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. 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 cyclic structures statically for RAGs. Such analysis would need to be conservative and it would be interesting to look into possible approaches and their applicability.

We also plan to *explore new application areas* for CRAGs, by looking into grammar flow analysis work [18,12].

Combining CRAGs with higher-order attribute grammars

Current work of other members of our research team includes adding support for higher-order attribute grammars (HAGs) [22]. Torbjörn Ekman is developing JastAdd in this direction. We believe that there are application areas where the *combination of CRAGs and HAGs* would prove useful. In the near future we hope to merge the HAG and the CRAG extensions in our tool JastAdd.

References

1. E. Bjarnason. *Interactive Tool Support for Domain-Specific Languages*. Licentiate thesis. Dept. of Computer Science, Lund University, Sweden, December 1997.
2. J. T. Boyland. *Descriptional Composition of Compiler Components*. Ph.D. thesis. University of California, Berkeley, 1996
3. A. Cornils, G. Hedin. Tool Support for Design Patterns based on Reference Attributed Grammars, *Proceedings of WAGA'00, Workshop on Attribute Grammars and Applications*. Ponte de Lima, Portugal. July 2000.
4. B. Courcelle, P. Franchi-Zannettacci. Attribute Grammars and Recursive Program Schemes. In *Theoretical Computer Science* 17, 163-191. 1982.
5. R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, pages 85-98. Palo Alto, California, July 1986. ACM. SIGPLAN Notices 21(7).
6. G. Hedin. An object-oriented notation for attribute grammars. *ECOOP'89*. BCS Workshop Series, pp 329-345, Cambridge University Press. 1989
7. G. Hedin. Reference Attributed Grammars, *Informatica* 24, 301-317. Slovenia 2000
8. F. Jalili. A general linear time evaluator for attribute grammars. *ACM SIGPLAN Notices, Vol 18(9):35-44, September 1983*.
9. JavaCC. http://www.experimentalstuff.com/Technologies/JavaCC/
10. L. G. Jones. Efficient evaluation of circular attribute grammars. *ACM Transactions on Programming Languages and Systems*, 12(3):429-462. 1990
11. M. Jourdan. An optimal-time recursive evaluator for attribute grammars. In M. Paul and B. Robinet, editors, *International Symposium on Programming, 6th Colloquium*, volume 167 of *Lecture Notes in Computer Science*, pages 167–178. Springer-Verlag, 1984.
12. M. Jourdan and D. Parigot. Techniques for Improving Grammar Flow Analysis. In Neil Jones, editor, *European Symposium on Programming (ESOP´90*). Lecture Notes in Computer Science 432, pp 240-255. Copenhagen, 1990. Springer-Verlag.
13. U. Kastens. Ordered Attributed Grammars. *Acta Informatica*, 13:229-256, 1980.
14. T. Katayama. Translation of attribute grammars into procedures. *ACM Transactions on Programming Languages and Systems*, 6(3):345-369. 1984
15. G. Kiczales et al. An Overview of AspectJ. In J. L. Knudsen, ed., *Proceedings of ECOOP 2001*, 327–353, Budapest, June 2001. LNCS 2072. Springer-Verlag.
16. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127-145, June 1968
17. O. L. Madsen. On defining semantics by means of extended attribute grammars. In *Semantics-Directed Compiler Generation*, pp 259-299, LNCS 94, Springer-Verlag, January 1980
18. U. Möncke and R. Wilhelm. Grammar flow analysis. In Henk Alblas and Borivoj Melichar, editors, *Attribute Grammars, Applications and Systems*. Lecture Notes in Computer Science 545, pp 151-186. Prague. June 1991. Springer Verlag.
19. P. Persson and G. Hedin. Interactive Execution Time Predictions Using Reference Attributed Grammars. In *WAGA'99, Second Workshop on Attribute Grammars and their Applications*. Amsterdam, The Netherlands, March 1999

20. A. Poetzsch-Heffter. Prototyping Realistic Programming Languages Based on Formal Specifications. *Acta Informatica* 34(10):737--772 (1997).

21. A. Sasaki, M. Sassa. Circular Attribute Grammars with Remote Attribute References. In *Waga´00, Third Workshop of Attribute Grammars and their Applications*. Ponte de Lima, Portugal. July 2000.

22. H. Vogt, S.Swierstra, M. Kuiper. Higher-Order Attribute Grammars. *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, Portland, Oregon. 1989.

Program Visualization using Reference Attributed Grammars

Eva Magnusson and Görel Hedin

Published in Nordic Journal of Computing 7 (2000).

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.

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 [1,2,3]. APPLAB is based on structure-oriented editing and reference attributed grammars [11] (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 [6] from University of Bremen as the visualization tool. See Fig. 1.

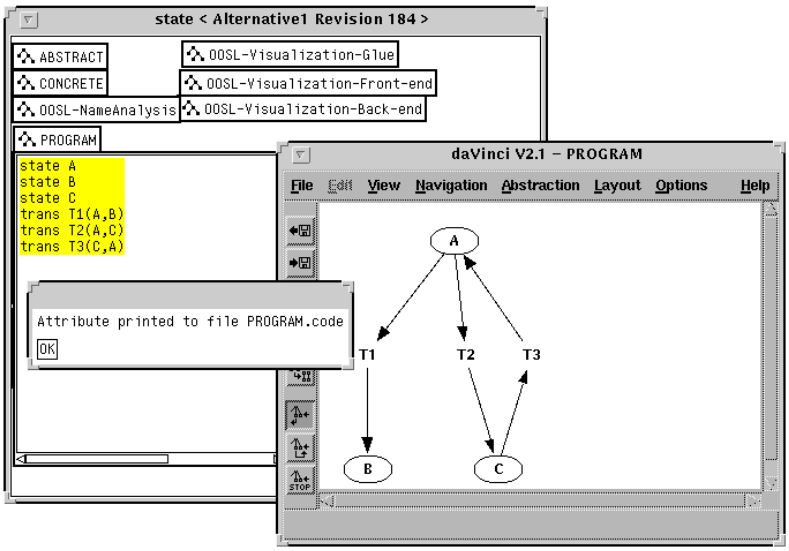


Figure 1 *Overall architecture. To the left, the language environment APPLAB with the language specification in several aspects (icons ABSTRACT, CONCRETE, OOSL-NameAnalysis, ...) and an example program in the specified language (window PROGRAM). To the right, the daVinci tool is used to visualize the 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) [1,2]. 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 resulting 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 [12] is an extension of a context-free grammar where a set of attributes A(X) are associated with each non-terminal sym-

bol 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...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,...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 commion in classical attribute grammars [9].

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 [14]. 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 [11]. 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 beeing defined declaratively by equations. This is in contrast to ususal 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 (NodeDic-

tionary) 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 [1,2,10]) 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 attrbute grammar systems such as the Synthesizer Generator [19]. From an object-oriented viewpoint, the OOSL modules are orthogonal to the class hierarchy. Similar modularization techniques are available also for some object-oriented languages, in particular the fragment system for the BETA language [13] and in subject-oriented programming [8].

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 [9]. The example also shows an example of an overriding equation: the equation in Program overrides the default definition in ANY-NODE 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 [6], 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 visualization 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.

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.

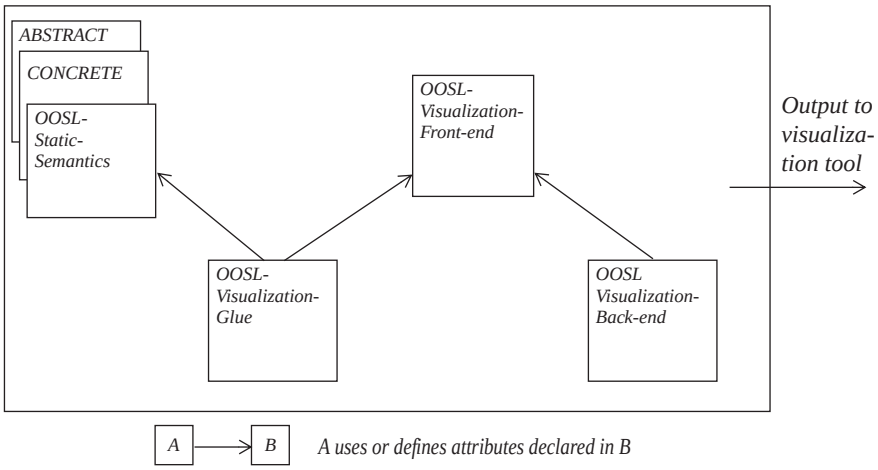


Figure 3 *Internal architecture in APPLAB showing module dependencies for a general reuasable specification of the visualization.*

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 production 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.

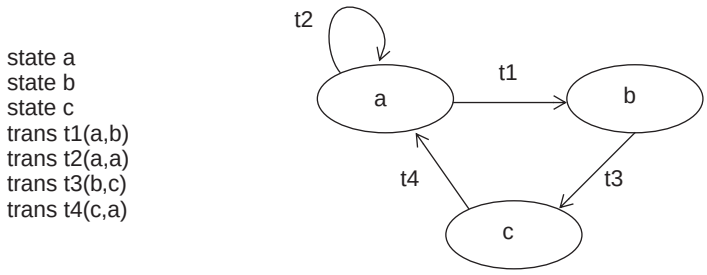


Figure 5 *A simple program and a possible visualization*

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 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 State-

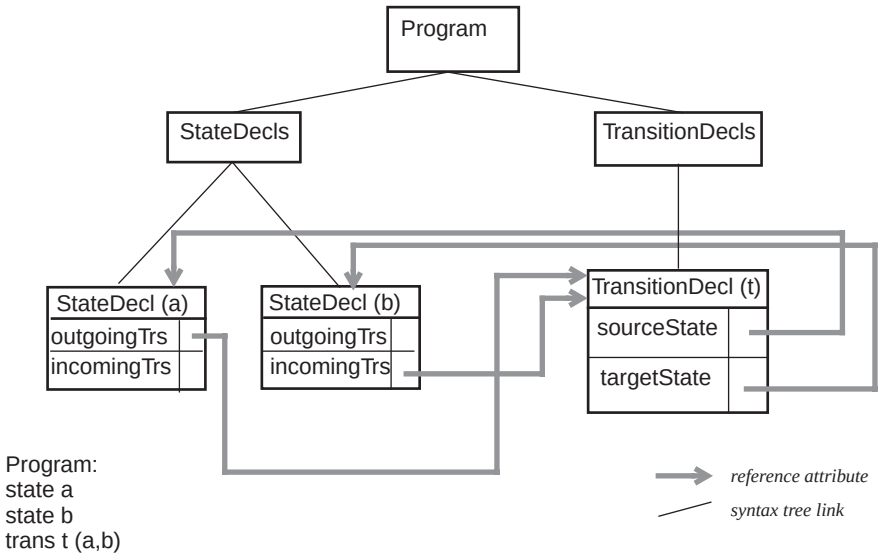


Figure 6 *Connections between states and transitions for a small program*

Decl 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 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.

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

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 defintion of* incomingTrs *is similar to* outTrs *and not shown.*

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 [16].

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*

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

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 int the* TransitionDecl *production. See Fig. 4.*

and targetState can be defined simply by doing a lookup in the dictionary stateDict 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.

4 Visualization back-end for a 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[child1, child2,...]. 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.

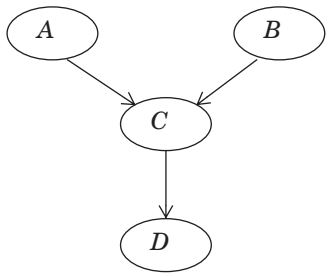


Figure 10 *A node (C) with more than one parent*

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 pre-

fixNbr 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 [16] 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 [16].

5 Reusing the visualization specification

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 [7]. 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.

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

```
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 DeInitalized {
        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*

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*

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.

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 lan-

guages. One example is Panorama, a visual environment for Java/C/C++ which supports visualizations like call graphs and flow diagrams [18]. Other examples include Rational Rose [21] and TogetherJ [24]. 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 [4]. 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 [20] 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 [5]. 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 [15] or anonymous inner classes in Java [17]. 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.

Acknowledgments

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.

References

1. E. Bjarnasson. APPLAB User's Guide. Technical Report, Dept. of Computer Science, Lund University, Sweden, September 1995.
2. E. Bjarnasson. Interactive Tool Support for Domain-Specific Languages. Licentiate thesis. Dept. of Computer Science, Lund University, Sweden, December 1997.
3. E. Bjarnasson, G. Hedin, K. Nilsson. Interactive Language Development for Embedded Systems. *Nordic Journal of Computing 6*, 1, 36-54. 1998.
4. P. T. Cox, F. R. Giles, T. Pietrzykowski. Prograph. In Visual Object-Oriented Programming. Manning Publications Co. pp 45-66. 1995.
5. J.H. Cross, T. D. Hendrix. Language Independent Program Visualization. In *Software Visualization*. Series on Software Engineering and Knowledge Engineering. Volume 7, pp 27-45. World Scientific, 1996.
6. daVinci V2.1 On-line Documentation. http://www.tzi.de/~davinci/docs/overviewF.html. 1998.
7. S. Gestegård. Emulation Software for Executable Specifications. Master's thesis. LU-CS-EX:99-6. Dept. of Computer Science, Lund University, Sweden, April 1999
8. W. Harrison, H. Ossherr. Subject-Oriented Programming - a Critique of Pure Objects. In *Proceedings of 1993 Conference on Object_oriented Programming Systems, Languages and Applications.*
9. G. Hedin. An Object-Oriented Notation for Attribute Grammars. ECOOP'89. BCS Workshop Series, pp 329-345, Cambridge University Press,1989.
10. G. Hedin. Incremental Semantic Analysis. PhD thesis, Dept. of Computer Science, Lund University, Sweden, March 1992.
11. G. Hedin. Reference Attributed Grammars. *Second Workshop on Attribute grammars and their Applications*. WAGA99, March 1999.
12. D.E.Knuth. Semantics of context-free languages. *Mathematics Systems Theory*, 2(2):127-145, June 1968.
13. B. B. Kristensen, O. L. .Madsen, B. M¢ller-Pedersen, K Nygaard. An Algebra for Program Fragments. In *Proceedings of ACM SIGPLAN 85 Symposium om Programming Languages and Programming Environments*. Seattle, Washington 1985.
14. O. L. Madsen. On defining semantics by means of extended attribute grammars. In *Semantics-Directed Compiler-Generation*. Springer-Verlag, 259-299, 1980.
15. O. L. Madsen, B. Møller-Pedersen. Part objects and their location. Proceedings of the 7th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 7). Dortmund, 1992. Prentice Hall.

16. E. Magnusson. State Diagram Generation using Reference Attributed Grammars. Technical Report. LU-CS_TR:2000-219. Dept. of Computer Science, Lund University, Sweden, March 2000.
17. Sun Microsystems. 1997. Inner classes in Java. http://www.java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc.html.
18. Panorama. A visual environment for Java/C/C++ software testing, quality assurance, documentation and maintenance.
Information available at http://www.softwareautomation.com. (1999).
19. T. Reps, T. Teitelbaum. *The Synthesizer Generator. A System for constructing language-based editors.* Springer-Verlag, 1989.
20. G-C Roman. Declarative Visualization. In *Software Visualization*. MIT Press, 173-186. 1998
21. Rational Rose. A visual modelling tool. Information available at http://www.rational.com/products/rose/index.jtmpl. (1999)
22. C. Roudet. Visualisation graphique incrémentale par evaluation d'attributs. Stage DEA informatique de l'ESSI, Université Nice, 1994. (http://www-rocq.inria.fr/oscar/www.fnc2/OpenPostscript/Roudet.ps.gz)
23. A. Schürr. PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems. Technical Report AIB 94-11, RWTH Aachen, Germany. 1994. (Available via http://www-i3.informatik.rwth-aachen.de/research/progres/main.html).
24. TogetherJ. Information available at http://www.togethersoft.com. (1999).

JastAdd—an aspect-oriented compiler construction system

Görel Hedin and Eva Magnusson

Published in SCP - Science of Computer Programming, 47(1). Elsevier. November 2002.

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.

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 [15]. A common alternative modularization technique is to use the Visitor design pattern [6,24]. 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) [9]. 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 [11]. 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 [4] 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 [5] and JTB [12] 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 [20], Jalili {10], and Jourdan [13]: 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 reference attribute grammars [9]. 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 [6].

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 [8], 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.
- *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).

```
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;
...
```

Figure 1 *Abstract grammar for Tiny*

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.

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.

```
 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*

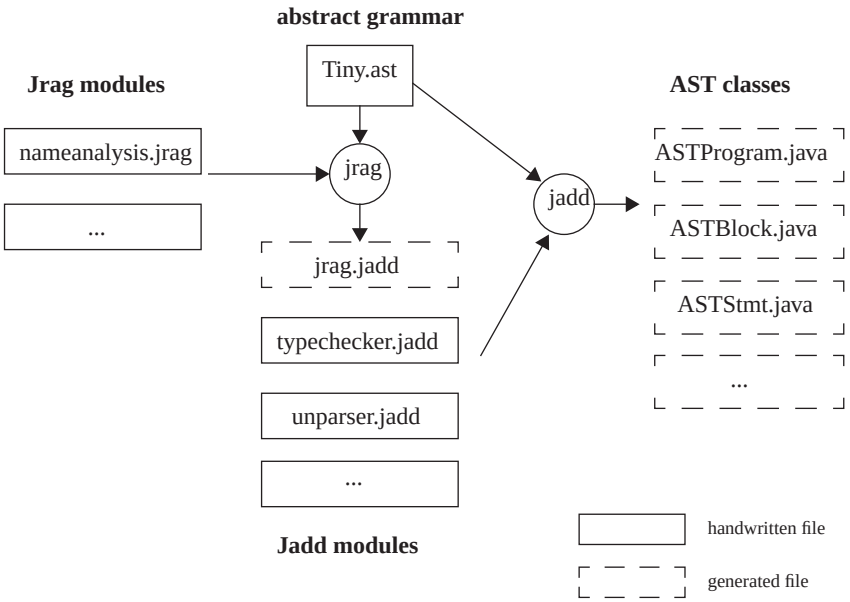


Figure 3 *Architecture of the JastAdd system.*

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

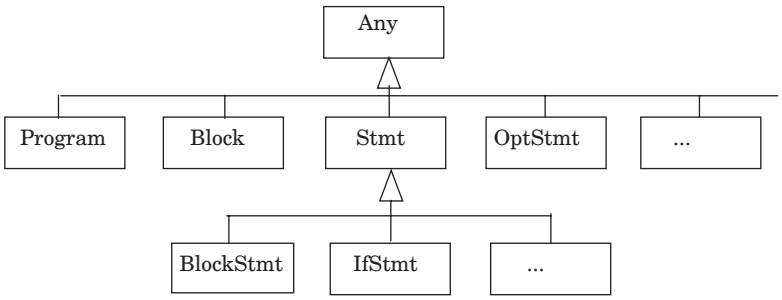


Figure 4 *Class diagram after adding the superclass Any.*

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.

2.5 Connection to the parser generator

2.5.1 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.

2.5.2 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 [6]. 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 [15].

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 ot 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. In the cur-

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

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 error");
    }
}
...
```

Figure 5 *Jadd modules for type checking 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 error");
        ...
    }
}
```

Figure 6 *Woven complete AST class*

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 anal-

ysis 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.

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 modules - 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.*

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 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 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

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 analsis.*

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  };
...
```

Figure 9 *A Jrag module for type checking*

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 differ-

ent 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 [10,13,20].

6 Related work

Recent developments in aspect-oriented programming [14] include the work on AspectJ [15], subject-oriented programming [7], and adaptive programming [19].

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 heavyweight 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 [17,16]. 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) [21] 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 [11], SableCC [5], Java Tree Builder [12], and JJForester [18]. 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 [3], and our own predecessing system Applab [2]. 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 [22]. We also plan to continue the development of reference attributed grammars and to applying them to new problem areas.

Acknowledgments

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.

References

1. ANTLR Translator Generator, http://www.ANTLR.org/.
2. Bjarnason, E., G. Hedin, K. Nilsson. Interactive Language Development for Embedded Systems. *Nordic Journal of Computing* 6(1):36–54 (1999).
3. Boyland, J. T. *Descriptional Composition of Compiler Components*. Ph.D. thesis. University of California, Berkeley, 1996.
4. CUP, LALR Parser Generator for Java, http://www.cs.princeton.edu/~appel/modern/java/CUP/
5. Gagnon, E. M., L. J. Hendren, *SableCC, an Object-Oriented Compiler Framework*. In Proceedings of Tools 26–USA'98. IEEE Computer Society (1998).
6. Gamma, E. et al., *Design Patterns*, Addison Wesley, 1995.
7. Harrison, W., H. Ossher, *Subject-Oriented Programming (A Critique of Pure Objects)*, OOPSLA 1993 Conference Proceedings. ACM SIGPLAN Notices, ACM Press, 28(10) (1993), 411–428.
8. Hedin, G., *An object-oriented notation for attribute grammars*, ECOOP'89. BCS Workshop Series, Cambridge University Press (1989), 329–345.
9. Hedin, G., *Reference Attributed Grammars*, Informatica (Slovenia) 24(3): (2000).
10. Jalili, F., *A general linear time evaluator for attribute grammars*, ACM SIGPLAN Notices, ACM Press, {\bf 18(9)} (1983), 35–44.

11. JavaCC, The Java Parser Generator, http://www.metamata.com/
12. JTB, Java Tree Builder, http://www.cs.purdue.edu/jtb/
13. Jourdan, M., *An optimal-time recursive evaluator for attribute grammars*. In M. Paul and B. Robinet, editors, International Symposium on Programming, 6th Colloquium, LNCS 167 (1984), 167–178. Springer Verlag.
14. Kiczales, G., et al. *Aspect-Oriented Programming*, ECOOP'97, LNCS 1241 (1997), 220–242. Springer Verlag.
15. Kiczales, G., et al. An Overview of AspectJ. In J. L. Knudsen, ed., *Proceedings of ECOOP 2001*, 327–353, Budapest, June 2001. LNCS 2072. Springer-Verlag.
16. Knudsen. J. L. Aspect-Oriented Programming in BETA using the Fragment System. In *Proceedings of the Aspect-Oriented Programming Workshop* at ECOOP'99.
17. Kristensen, B. B., et al. Syntax-Directed Program Modularization. In P. Degano, E. Sandewall (eds.): *Integrated Interactive Computing Systems*, North-Holland Publishing Company, 1983.
18. Kuipers T., Visser J. Object-oriented tree traversal with JJForester. In *Proceedings of LDTA'0*1. Genova, Italy, April 2001. Electronic Notes of Theoretical Computer Science, Elsevier
19. Lieberherr, K., *Adaptive Object-Oriented Software*, PWS Publishing Company, 1996.
20. Madsen, O. L. *On defining semantics by means of extended attribute grammars. In* Semantics-Directed Compiler Generation, LNCS 94 (1980), 259–299. Springer Verlag.
21. Madsen, O. L., C. N¢rgaard. *An Object-Oriented Metaprogramming System*. In proceedings of Hawaii International Conference on System Sciences 21, (1988).
22. de Moor, O., S. Peyton-Jones, E. van Wyk. Aspect-Oriented Compilers. In *Generative and Component-Based Software Engineering, First International Symposium*. LNCS 1799, (1999), 121–133. Springer Verlag.
23. Poetzsch-Heffter, A. Prototyping Realistic Programming Languages Based on Formal Specifications. *Acta Informatica* 34(10):737–772 (1997).
24. Watt, D. A., D. F. Brown. *Programming Language Processors in Java*, Prentice Hall, 2000.
25. Woolf, B. The Null Object Pattern. In R. Martin et al. (eds.): *Pattern Languages of Program Design*, Addison-Wesley, 1997.

Circular Reference Attributed Grammars – their Evaluation and Applications

Eva Magnusson and Görel Hedin

To appear in ENTCS, Electronic Notes of Theoretical Computer Science, Elsevier.

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.

1 Introduction

Attribute grammars (AGs), as introduced by Knuth [15], 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) [7] and similar formalisms, e.g., [19], [2]. 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 [7], execution time prediction [18], program visualization [17], and design pattern checking [4].

The second of the restrictions mentioned above, circular definitions, is lifted by Circular Attribute Grammars (CAGs) such as those of Farrow [5] and Jones [11]. The traditional AG requirement of non circularity 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., [5,11,20]) 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 [5] 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 Reference Attributed Grammars (RAGs) and Circular Attribute Grammars (CAGs) 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 [8], 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 hand-coded iterative implementation.

There is some previous work on combining RAG-like formalisms with CAGs. Boyland has also suggested this general combination, but the evaluation algorithm presented in his thesis is unclear [2]. Sasaki & Sassa present Circular Remote Attribute Grammars (also abbreviated CRAGs), which on the surface is similar to our CRAGs [20]. 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 optimal evaluation is achieved by applying the semantic functions in that order.

Traditional AGs are required to be noncircular, but, as has been shown by, e.g., Farrow [5] and Jones [11], 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. 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.

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 Evaluation of circular attribute grammars

Jones [11] 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 [5] introduced a static evaluation technique based on the one by Katayama [14], 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 [20] 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.

2.2 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 [16], Jalili [9], and Jourdan [12], although we use an object-oriented implementation [8]. 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. The evaluator is implemented in Java

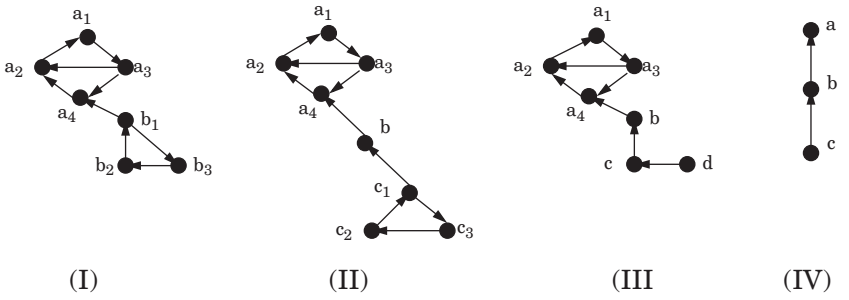


Figure 2 *Different cases of dependencies involving strongly connected components.*

which provides a straight-forward implementation of the algorithm. Fig. 3 shows a fragment of an AG 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 raises an exception 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 [8] for details.

2.3 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 [8]. 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.

RAG	*Evaluator code*

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

Block: Any ::= Decl* Stmt* {
Decl lookup(String name) { ...}
sons Any.env = this; }
	class Block **extends** Any {
	... 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 a RAG and corresponding evaluator*

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 and its default equation. Equations may be overridden in subclasses, e.g., the equation for env is overridden in Block. The **sons** clause used in these equations means that the equation applies to all sons of a specific type. 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 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 2.2 and 2.3 to handle CRAGs. Fig. 6 shows a basic evaluation algorithm for the circular attribute s.

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 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 to fulfil 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 2.1 are avoided by our basic algo-

rithm 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 2.1. 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 non circularly 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.

```
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*

Avoiding iterative evaluation of actually noncircular attributes

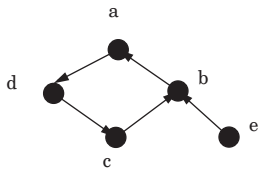
For many ASTs there might be many actually noncircular instances of potentially circular attributes. Consider case (IV) in Fig. 2 and suppose a is demanded. If a is potentially circular, an iterative process is started in which b and c will be demanded. Again, a small modification of the algorithm makes 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 [20] 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



a = b
b = c.union(e)
c = d
d = a
e = {id}

Figure 8 *Equations for some attributes creating cyclic dependency*

attribute c as **circular** and suppose that attribute a is demanded. An iterative process is started, b is demanded and then c i 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.

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. This is in contrast to ordinary attribute grammars where 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 [13], 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 [5]. The development of static evaluators for subcategories of RAGs and CRAGs is an open problem that we have not pursued, but there is some other work in this direction. In [3] Boyland addresses the problem of analyzing (noncircular) non-local dependencies statically. The scope and implementation status of the developed technique is, however, unclear. Sasaki & Sassa [20] 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 [11] is optimal. In section 3.3 we showed how our dynamic demand-driven algorithm avoids some of the suboptimal cases by using cached attribute values. For non-circular RAGs, dynamic demand-driven evaluation using caching is optimal (each attribute is evaluated at most once). For CRAGs, the existence of general optimal evaluation algorithms is an open issue.

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.

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 [5] and means that the 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 [1]).

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 → X Y Z | d	false	{a, c, d}	∅

Figure 9 *Example CFG and its values for nullable, first, and follow*

4.1.1 Computation of *nullable*

We define *nullable* by the following equations:

(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(γ_1) || nullable(γ_2) ... || nullable(γ_n)

(ii) Let ε be an empty sequence of terminal and nonterminal symbols.
The empty sequence is nullable:
nullable(ε) == 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(γ) == nullable(*s*) && nullable(δ)

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

The definition 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 of one of the productions.

The above definition can, with trivial adaptions to syntax form, be formulated directly in a CRAG as demonstrated in Fig. 10. The CRAG equations corresponding to equations (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 2.3. Their values for nullable are equal as indicated by equation (v).

```
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*

4.1.2 Computation of *first*

The following equations define 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(γ_1) $\cup$ first(γ_2) ... $\cup$ first(γ_n)

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

(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(δ)
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 of one of the productions. We can also note that the definition of *first* relies on the definition of *nullable*. Figure 11 shows the corresponding CRAG including the equations (i) - (iv) from the definition 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*

4.1.3 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 attributes are straight-forward using CRAGs, for example by defining a set of NUse references at each NDecl. With these attributes in place, the specification of *follow* becomes as straight-forward as for *nullable* and *first*.

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.

4.1.4 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 [1]. 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 [10]. 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. [1] 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.

			CRAG			**Handwritten**
Language	**#N**	**#P**	**#I-T**	**#I-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.*

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.

4.2 Using constants before declaration

This is an example described by Farrow in [5] 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 2.3. 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.

The val attribute will be noncircular exactly when Farrow's cyclic specification produces a complete table of constant values, i.e., when each con-

```
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 *A CRAG for a language where constants can be used before declaration.*

stant 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 isCircular 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 if the constant is itself in this set. An alternative way is to specify the dependency sets of each constant using a circularly specified attribute of a set type and then check if a constant is contained in its own dependency set.

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 [5], Jones [11], and Sasaki & Sassa [20] 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 and the evaluation technique used (or rather the technique to check for convergence) is dependent on this fact. This means that the evaluation process described by Sasaki & Sassa in [20] would not be capable of handling, e.g., a language with structured loop-statements. 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 where the fixed point is 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.

Open problems concerning RAGs that are interesting for future research include whether it is possible to detect potential cyclic dependencies statically and to what extent it is possible to statically analyze dependencies in general. The latter question can be addressed as in [20] 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 eval-

uator are unnecessary. It would be useful if algorithms for a more realistic analysis could be developed at least for some subcategories of RAGs.

Future work also 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. 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 [20]. Finally, we plan to apply CRAGs to more problem areas.

References

1. A. W. Appel. Modern Compiler Implementation in Java. Cambridge University Press. 1998.
2. J. T. Boyland. Descriptional Composition of Compiler Components. Ph.D-thesis. University of California, Berkeley, California. 1996.
3. J.T. Boyland. Analyzing Direct Non-Local Dependencies in Attribute Grammars. In *Proceedings of CC'98: International Conference on Compiler Construction*, pp 31-49. LNCS 1383, Springer-Verlag. 1998.
4. A. Cornils, G. Hedin. Tool Support for Design Patterns based on Reference Attributed Grammars, *Proceedings of WAGA'00, Workshop on Attribute Grammars and Applications*. Ponte de Lima, Portugal. July 2000.
5. R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, pages 85-98. Palo Alto, California, July 1986. ACM. SIGPLAN Notices 21(7).
6. G. Hedin. An object-oriented notation for attribute grammars. *ECOOP'89*. BCS Workshop Series, pp 329-345, Cambridge University Press. 1989.
7. G. Hedin. Reference Attributed Grammars, *Informatica* 24, 301-317. Slovenia 2000.
8. 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.
9. F. Jalili. A general linear time evaluator for attribute grammars. *ACM SIGPLAN Notices, Vol 18(9):35-44, September 1983*.
10. JavaCC. http://www.webgain.com/products/java_cc/
11. L. G. Jones. Efficient evaluation of circular attribute grammars. *ACM Transactions on Programming Languages and Systems*, 12(3):429-462. 1990
12. M. Jourdan. An optimal-time recursive evaluator for attribute grammars. In M. Paul and B. Robinet, editors, *International Symposium on Programming, 6th Colloquium*, volume 167 of *Lecture Notes in Computer Science*, pages 167–178. Springer-Verlag, 1984.
13. U. Kastens. Ordered Attributed Grammars. *Acta Informatica*, 13:229-256, 1980.
14. T. Katayama. Translation of attribute grammars into procedures. *ACM Transactions on Programming Languages and Systems*, 6(3):345-369. 1984
15. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127-145, June 1968.
16. O. L. Madsen. On defining semantics by means of extended attribute grammars. In *Semantics-Directed Compiler Generation*, pp 259-299, LNCS 94, Springer-Verlag, January 1980.
17. E. Magnusson, G. Hedin. Program Visualization using Reference Attributed Grammars, *Nordic Journal of Computing* 7(2000) 67-86.

18. P.Persson and G. Hedin. Interactive Execution Time Predictions Using Reference Attributed Grammars. In *WAGA'99, Second Workshop on Attribute Grammars and their Applications*. Amsterdam, The Netherlands, March, 1999.
19. A. Poetzsch-Heffter. Prototyping realistic programming languages based on formal specifications. *Acta Informatica* 34 (1997), 737-772.
20. A. Sasaki, M. Sassa. Circular Attribute Grammars with Remote Attribute References. In *Waga´00, Third Workshop of Attribute Grammars and their Applications*. Ponte de Lima, Portugal. July 2000.