# Reference Attributed Grammars

GÖREL HEDIN

*Department of Computer Science, Lund University, Sweden*
*email: Gorel.Hedin@cs.lth.se*

**Abstract**

An extension to canonical attribute grammars is introduced, permitting attributes to be references to arbitrary nodes in the syntax tree, and attributes to be accessed via the reference attributes. Important practical problems such as name and type analysis for object-oriented languages can be expressed concisely in these grammars, and an optimal evaluation algorithm is available. The proposed formalism and algorithm have been implemented in an interactive language development tool.

## 1 Introduction

Canonical attribute grammars (AGs), as introduced by Knuth [20], is an appealing formalism that allows context-sensitive properties of individual constructs in a language to be described in a declarative way, and to be automatically computed for any program in the language. Important applications include defining context-sensitive syntax and code generation for a language.

A major problem with canonical AGs is that the specifications often become too low-level when dealing with non-local dependencies, i.e., situations where a property of one syntax tree node is dependent on properties of nodes far away in the tree. For example, the type of an identifier use site depends on the type of the declaration which may be located arbitrarily far away in the tree.

Many researchers have suggested different extensions to attribute grammars to solve this problem, e.g. [3, 4, 10, 11, 13, 14, 17, 27]. We propose yet another extension: Reference Attributed Grammars (RAGs). Advantages of RAGs over previous approaches include that they can handle advanced name and type analysis problems (including for object-oriented languages) without the need for circular dependencies, and also that there is an optimal evaluation algorithm that works for any non-circular RAG. We have implemented the formalism and evaluation algorithm in our interactive language development tool APPLAB (APPlication language LABoratory) [5, 6]. The key idea in RAGs is to allow attributes to be references denoting nodes arbitrarily far away in the syntax tree. Attributes of other nodes can be accessed via such reference attributes.

The rest of this paper is structured as follows. In Section 2 a background is given on canonical AGs and their drawbacks. Section 3 introduces RAGs, discusses the evaluation algorithm, and compares RAGs to canonical AGs. Section 4 adds some object-oriented extensions to RAGs, including a class hierarchy for non-terminals and support for virtual function attributes. Section 5 shows an extensive example of name and type analysis for an object-oriented language, PicoJava. Section 6 discusses our tool APPLAB, section 7 relates to other work, and section 8 concludes the paper and suggests future research.

## 2 Background

### 2.1 Canonical attribute grammars

A canonical attribute grammar consists of a context-free grammar extended with attributes for the non-termi-

nals and semantic rules for the productions. The attributes are characterized as *synthesized* or *inherited*, depending on if they are used to transmit information downwards or upwards in the syntax tree. Given a production $X_0 \rightarrow X_1 \ldots X_n$, a semantic rule is written $\alpha_0 = f(\alpha_1,\ldots,\alpha_m)$ and defines $\alpha_0$ as the value of applying the *semantic function* $f$ to the attributes $\alpha_1,\ldots,\alpha_m$. The attribute $\alpha_0$ must be either a synthesized attribute of $X_0$ or an inherited attribute of $X_j$, $1 \le j \le n$. I.e., a semantic rule defines either a synthesized attribute of the left-hand symbol of the production, or an inherited attribute of one of the symbols on the right hand side of the production. A function argument, $\alpha_k$, $1 \le k \le m$, must be an attribute of $X_j$, $0 \le j \le n$. I.e., a rule is local, depending only on information available in the attributes of the symbols of the production.

A grammar is considered to be *well-formed* if each attribute in any syntax tree of the grammar has exactly one defining semantic rule. This is obtained by restricting the start symbol to have synthesized attributes only, and by requiring a production $X_0 \rightarrow X_1 \ldots X_n$ to have exactly one rule for each synthesized attribute of $X_0$ and one rule for each inherited attribute of $X_j$, $1 \le j \le n$.

The assignment of values to attributes of a syntax tree is called an *attribution*. An attribution is called a *solution* if all semantic rules are satisfied. A well-formed grammar is considered to be *well-defined* if there exists exactly one solution (or one *best* solution according to some criteria) for each syntax tree of the grammar.

If an attribute $a_1$ is used for defining another attribute $a_2$ we say that there is a *dependency* ($a_1$, $a_2$). If the dependency graph for a syntax tree is non-circular, the attribution can be obtained simply by applying the semantic functions in topological order, provided that the semantic functions terminate. If any syntax tree derivable from a grammar will have a non-circular dependency graph, the grammar is said to be *non-circular*. Usually, canonical AGs are required to be non-circular, but there are also extensions which allow circular dependencies. The usual requirement for such grammars is that the values in the domain of an attribute on a cyclic dependency chain can be arranged in a lattice of finite height, and that all semantic functions are monotonic with respect to these lattices. In this case, there will be at least one solution, and the solution with the "least" attribute values is taken to be the best one. For such non-circular grammars, the attribution can be obtained by iteratively applying the semantic functions, giving the attributes on the cycle the lattice bottom values as start values. See, e.g. [21, 7, 15].

### 2.2 Problems with canonical attribute grammars

Canonical AGs are well-suited for description of problems where the dependencies are local and follow the syntax tree structure. For example, in type analysis, the type of an operator typically depends on the types of its operands. Canonical AGs are less suited for description of problems with non-local dependencies, such as name analysis problems where properties of an identifier use site depends on properties of an identifier declaration site. Typically, the use and declaration sites can be arbitrarily far away from each other in the tree, and any information propagated between them needs to involve all intermediate nodes. There are several drawbacks with this.

One drawback is that the information about declarations in the syntax tree needs to be replicated in the attributes: To do static semantic analysis, all declared names in a scope, together with their appropriate type information, need to be bundled together into an aggregate attribute, the "environment", and distributed to all potential use sites. At each use site, the appropriate information is looked up.

A second drawback is that the aggregate attributes with information replicated from the syntax tree can become very complex. The distribution of the aggregate information works well for procedural languages with Algol-like scope rules (nested scopes), but is substantially more difficult for languages with more complex scope rules, for example modular languages and object-oriented languages. For example, the use of qualified access in a language implies that it is not sufficient with a single environment attribute at each use site - it is necessary to provide access to all potentially interesting environments and select the appropriate one depending on the type of the qualifying identifier. The aggregate attributes thus need to become more complex, and to contain also information about relations between different declarations. The semantic functions working on these complex attributes naturally also become more complex. The AG formalism does not itself support the description of these complex attributes and functions.

A third drawback is that it is difficult to extend the grammar. Suppose we have a grammar with a working name analysis for extracting types, and we want to extend it by propagating also the declaration mode, i.e. information about if the declaration is a constant or a variable. There are two alternatives for modelling this. Either we introduce an additional environment attribute which maps names to modes and is defined analogously to the environment mapping names to types. Just like the type environment, the definition of the mode environment needs to involve all intermediate nodes. A second alternative is to modify the original type environment to also include mode information. None of these alternatives is very attractive since we cannot describe the extension in a clean concise way.

A fourth drawback with canonical grammars is that they are not suited for incremental evaluation. This is partly because there is no mechanism for incremental updating of the aggregated attributes (environments) and partly because a change to a declaration typically affects attributes all over the syntax tree (i.e., the environments), even though the extracted information is unchanged. Incremental evaluation based on this model does thus not scale up.

In this paper we address the first three of these drawbacks.

## 3 Reference Attributed Grammars (RAGs)

### 3.1 Reference Attributes

Canonical attribute grammars assume value semantics for the attributes. I.e., an attribute cannot (conceptually) be a reference to an object, or have a value containing such references. From an implementation point of view it is possible, and common, to implement two attributes with the same value as references to the same object. However, this is merely an implementational convenience for saving space, and the fact that these two attributes refer to the same object cannot be used in the grammar. I.e., the implementation is referentially transparent, preserving the value semantics of the grammar.

In our extension to canonical attribute grammars, attributes are allowed to be references to nodes in the syntax tree. Thus, we abandon the value semantics and introduce reference semantics. Structured attributes like sets, dictionaries, etc., may also include reference values. As we will show in Section 5, the use of reference values makes attribute grammars well-suited for expressing problems with non-local dependencies that do not necessarily follow the syntax tree structure.

A reference value denoting a node in the syntax tree may be dereferenced to access the attributes of that node. This way, a reference attribute constitutes a direct link from one node to another node arbitrarily far away in the syntax tree, and information can be propagated directly from the referred node to the referring node, without having to involve any of the other nodes in the syntax tree. We call an attribute grammar extended with this capability a *reference attributed grammar* (RAG).

### 3.2 TINY: an example RAG

Figure 1 shows the RAG specification of TINY, a tiny language made up to illustrate some central concepts in RAGs. TINY is so simple that it has only one possible syntax tree, which is shown with its attribution in Figure 2.

The example illustrates important aspects of RAGs. First, by considering the reference attributes in addition to the tree links, the syntax tree can be viewed as a (syntax) graph. The syntax graph may contain cycles: the B node contains a reference attribute rC denoting the C node which in turn contains a reference attribute rB referring back to the B node. However, although the syntax graph contains a cycle, the dependencies between the attributes form a non-circular graph, and the RAG is thus non-circular. Since all semantic functions terminate, the RAG is well-defined, and a unique solution has been found for the tree by evaluating the attributes in topological order, e.g., rB, c, rC, b.

The value of a reference attribute is the (unique) identity of the denoted node, drawn as an arrow in the figure. This value can be computed before the attributes of the denoted node are evaluated, and does thus not

| *Nonterminal* | *Attributes* | *Productions* | *Semantic rules* |
|---|---|---|---|
| A | | A → B C | B.rC = C<br>C.rB = B |
| B | ↓ rC: **ref** (C)<br>↑ b: **integer** | B → | B.b = B.rC.c |
| C | ↓ rB: **ref** (B)<br>↑ c: **integer** | C → | C.c = 7 |

**Fig. 1.** RAG specification of TINY

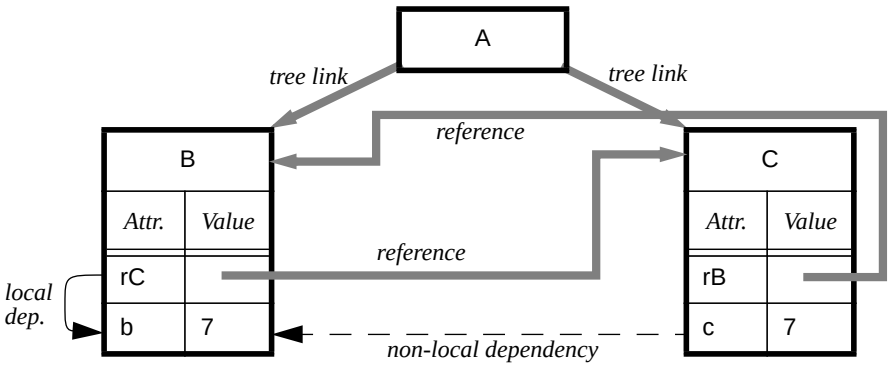


**Fig. 2.** RAG attribution of TINY (non-circular)

depend on those attributes. In the example, the semantic rules defining rC and rB depend only on constant values (the identities of nonterminals B and C), and rB and rC do therefore not have any incoming dependency edges.

In a canonical AG all dependencies are local, i.e., they occur because an attribute of a non-terminal $X_1$ in a production is defined using an attribute of a non-terminal $X_2$ in the same production. For any given syntax tree, it is possible to determine the complete dependency graph without evaluating any attributes. In a RAG, there are non-local dependencies in addition to the local dependencies. A non-local dependency occurs between an attribute *a* defined by a semantic function and another attribute *b* whose value is obtained by qualified access via a reference attribute *r*. The dependency (*b*,*a*) can be determined only after evaluating the reference attribute *r*. In the TINY example, the non-local dependency from c to b can be determined only after rC has been given a value.

As will be shown in section 5, practical grammars for complex problems, like name analysis for object-oriented languages, can be written concisely using a non-circular RAG.

### 3.3 Attribute evaluation

Similar to a non-circular canonical AG, a non-circular RAG can be evaluated simply by following the dependencies, evaluating the attributes in topological order. As noted above, the dependency graph for a RAG cannot, in contrast to canonical AGs, be completely determined before evaluation, it has to be determined *during* the evaluation. Algorithms based on static computation of dependency graphs, such as for OAGs [18] are therefore not immediately applicable to RAGs. However, demand-driven algorithms, i.e., where each attribute

access is replaced by a call to the corresponding semantic function, can be directly used for RAGs and will work for any non-circular RAG. By caching an attribute value at the first access and returning the cached value at subsequent accesses, this evaluation algorithm becomes optimal. Several implementations of this algorithm have been presented for canonical attribute grammars [21, 12, 16]. In our system (APPLAB), we have implemented the algorithm for RAGs by using techniques from object-oriented programming, as described for canonical AGs in [8]. This technique fits well with the object-oriented extensions we have done to RAGs (see Section 4) and makes the translation particularly simple.

### 3.4 Translation of a RAG to a canonical AG

To show the relation between a RAG and a canonical AG we will discuss two different ways a RAG can be translated into a canonical (but in general circular) AG: table translation and substitution translation.

**Table translation**

In *table translation*, the idea is to model references as indices into a large table, with one entry per node in the syntax tree, and where each entry contains the attributes of the respective node. This table can itself be described as an attribute and be made available throughout the syntax tree so that dereferencing a reference attribute can be replaced by indexing into the table. The table translation will lead to a circular AG, but which may still be well-defined and possible to evaluate with iterative methods. The detailed steps of the table translation are as follows.

- For each symbol $X$ in the grammar, an attribute id is defined in such a way that the id attributes enumerate the nodes in the syntax tree in a preorder traversal. I.e., the root will have id = 1, its leftmost son id = 2, and so on. To define id, a help attribute maxId is introduced which contains the maximum id used in the subtree of $X$.
- An attribute ct (the "contents") is defined for each symbol $X$ as a tuple $<a_1, .., a_k>$ where $a_1$ .. $a_k$ are the original attributes in $X$. The $i$'th field in the tuple can be accessed by the notation ct($i$).
- An attribute allCt is defined for each symbol $X$ as an array of size $|T|$, where allCt[$n$.id]=$n$.ct for any node $n$ in the syntax tree $T$. To define allCt, array slices are collected bottom up using a synthesized attribute subCt. The allCt attribute is equal to subCt of the root, and that value is propagated down to each node using inherited allCt attributes.
- Each reference attribute $r$ is replaced by an integer attribute $r$.
- In semantic rules, an access to a symbol $X$ (used as a reference value) is replaced by the expression $X$.id, i.e. the id attribute of the $X$ node.
- In semantic rules, a dereferencing expression $r.a$, where $r$ is a reference denoting a node of nonterminal $X$ and $a$ is an attribute of the denoted node, is replaced by the expression allCt[$r$]($i$), where $a$ is the $i$:th attribute of $X$.

While this translation is straight-forward, it introduces circular attribute dependencies which are not allowed in canonical attribute grammars. In particular, any attribute $a$ defined using attribute dereferencing introduces a circular dependency since it depends on $T$, and the definition of $T$ in turn depends on $a$. However, although the translated grammar is in general circular, it is well-defined (provided that the RAG is non-circular), and possible to evaluate using iterative algorithms.

Figure 3 shows the specification of TINY, translated by table translation to canonical AG form. Figure 4 shows the resulting syntax tree and its attribution solution (some values are left out for brevity). The dereferencing of the reference attribute rC leads to a circular dependency chain. However, the grammar is well-defined: a unique solution has been found for the tree.

| | *Attributes* | | *Semantic rules* |
|---|---|---|---|
| A | ↑ id: **integer**<br>↑ ct: <><br>↑ subCt: **array**[**tuple**]<br>↑ allCt: **array**[**tuple**] | A → B C | B.rC = C.id<br>C.rB = B.id<br>A.id = 1<br>B.id = A.id+1<br>C.id = B.maxId+1<br>A.ct = <><br>A.subCt = [A.id → A.ct] ∪ B.subCt ∪ C.subCt<br>A.allCt = A.subCt<br>B.allCt = A.allCt<br>C.allCt = A.allCt |
| B | ↓ rC: **integer**<br>↑ b: **integer**<br>↓ id: **integer**<br>↑ maxId: **integer**<br>↑ ct: <**integer**, **integer**><br>↑ subCt: **array**[**tuple**]<br>↑ allCt: **array**[**tuple**] | B → | B.b = B.allCt[B.rC](2)<br>B.maxId = B.id<br>B.ct = <B.rC, B.b><br>B.subCt = [B.id → B.ct] |
| C | ↓ rB: **integer**<br>↑ c: **integer**<br>↓ id: **integer**<br>↑ maxId: **integer**<br>↑ ct: <**integer**, **integer**><br>↑ subCt: **array**[**tuple**]<br>↑ allCt: **array**[**tuple**] | C → | C.c = 7<br>C.maxId = C.id<br>C.ct = <C.rC, C.c><br>C.subCt = [C.id → C.ct] |

**Fig. 3.** Table-translated specification of TINY (canonical AG form)

**The substitution translation**
An alternative to the table translation is to translate RAGs by replacing each reference attribute by the corresponding ct attribute, i.e. the tuple containing the attributes of the denoted syntax node. In this translation, the allCt attribute is not needed. We refer to this translation method as the *substitution translation*. The problem with this method is that if a reference attribute is part of a circular data structure, it will have an infinite value in the translated canonical AG, and also give rise to a circular dependency chain. Figure 5 shows the attribution for TINY for such a translation. We might consider a refinement of this method where ct would include only the subset of attributes that are accessed via references. For TINY, such a translation would yield a non-circular canonical AG. However, there are other non-circular RAGs for which such a refinement will still produce a circular AG with infinite attribute values. Consider, e.g., extending C with an attribute d = rB.b.

## 4 Object-Oriented RAGs

In this section, we will introduce some extensions to RAGs which make specifications more concise. These extensions are based on an object-oriented view of attribute grammars, where nonterminals are viewed as superclasses and productions as subclasses. We refer to such extended grammars as *object-oriented reference attributed grammars* (ORAGs). ORAGs extend RAGs with virtual function attributes and an extended class hierarchy of nonterminals.

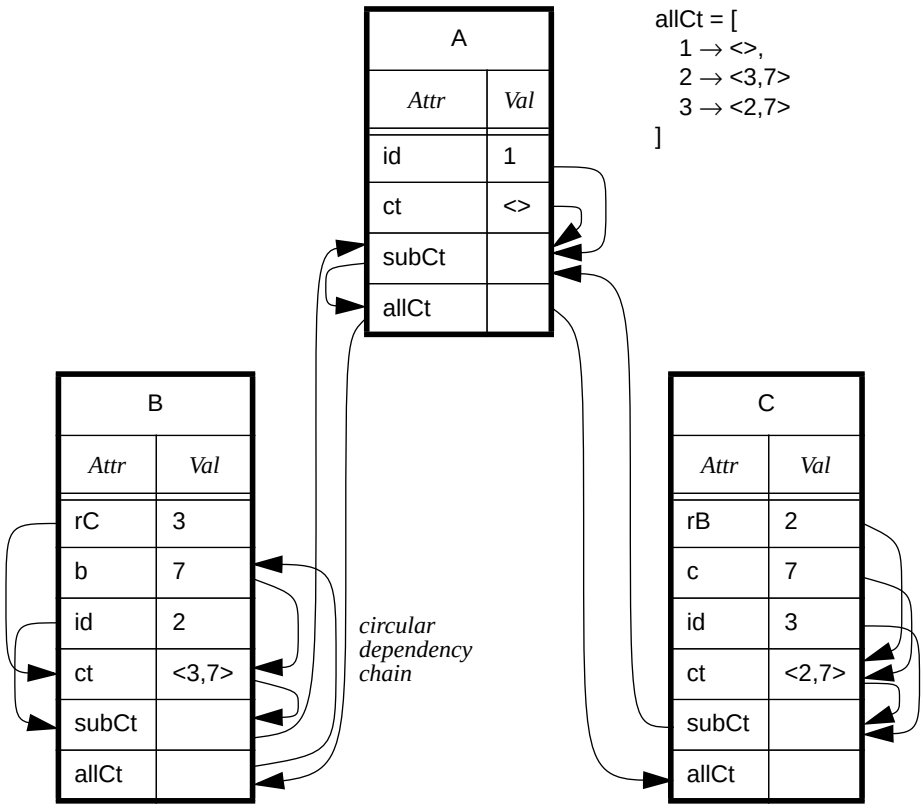


**Fig. 4.** Attribution of TINY for table-translated specification (circular)

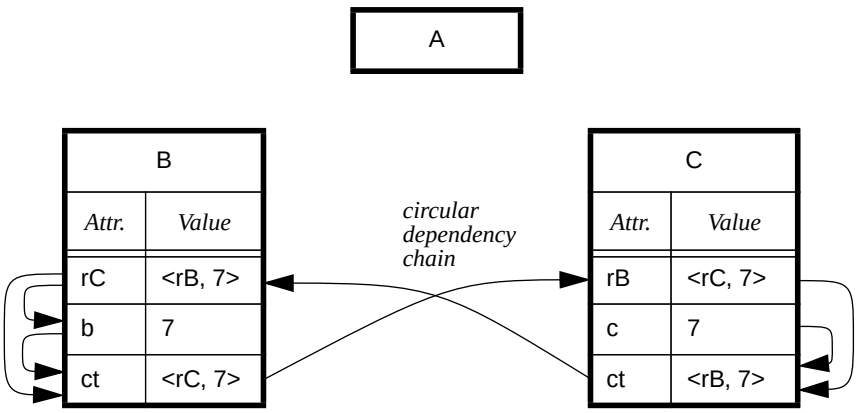


**Fig. 5.** Attribution of TINY for substitution-translated specification (circular). Attributes rC and rB have infinite attribute values

### 4.1 Virtual function attributes

Canonical AGs have a straight-forward translation to object-oriented programming [8]. In particular, a synthesized attribute is equivalent to a parameterless virtual function: The declaration of a synthesized attribute *a* of a nonterminal *X* is modelled by a declaration of a virtual function *a*() in a class *X*; and a semantic rule defining *a* in a production *p* is modelled by a virtual function implementation in a class *p* which is a subclass of *X*.

With this view, it is close at hand to make a generalization: to allow virtual functions *with* parameters. However, for a canonical AG, such a generalization is not necessary. This is because the number of accesses to an attribute is always bounded, so if parameters are desired, they can be modelled by inherited attributes. For RAGs, the situation is different. Because of the reference attributes, there may be an unbounded number of accesses to a given attribute. For example, in a typical RAG an identifier use site has a reference attribute denoting the appropriate declaration node. Since a declaration can be used in an unbounded number of places in the syntax tree, the number of references to a given declaration node, and thereby also the number of accesses to attributes in the declaration node, is not bounded by the grammar. In RAGs, parameters to virtual functions can therefore not be modelled by inherited attributes.

In ORAGs, we generalize synthesized attributes by allowing nonterminals to have *virtual function attributes*. A virtual function attribute $v(b_1,\ldots,b_k)$ of a nonterminal $X_0$, is similar to a synthesized attribute in that it must be defined by a semantic rule of each production $X_0 \rightarrow X_1 \ldots X_n$. A semantic rule for $v(b_1,\ldots,b_k)$ is written $v(b_1,\ldots,b_k) = f(b_1, \ldots, b_k, a_1, \ldots, a_m)$, where $a_i$, $1 \le i \le m$, is an attribute of $X_j$, $0 < j < n$. From this we see that a virtual function attribute *w*(), i.e. a parameterless virtual function, is equivalent to a synthesized attribute.

A virtual function attribute can be translated to RAG form by replacing each semantic rule by an auxiliary function, and making use of type casing to call the correct function. This translation is analogous to translating object-oriented programs to procedural programs.

### 4.2 Extended class hierarchy

The object-oriented view on attribute grammars gives a two-level class hierarchy where nonterminals are viewed as superclasses, i.e. general concepts, and productions as subclasses, i.e. specialized concepts. Taking this view, it is natural to expand the class hierarchy into more levels. In doing this we differ between *abstract nonterminals* and *concrete nonterminals*. An abstract nonterminal differs from a concrete nonterminal in that it may not occur in any production and it may not have a concrete nonterminal as its superclass. Abstract nonterminals are thus irrelevant for the context-free part of the grammar. They are introduced in order to simplify the description of the attribution, allowing common behavior (in the form of attributes and semantic rules) to be factored out. They are also useful as types for reference attributes.

We make use of a rooted single-inheritance class hierarchy, i.e. each nonterminal has exactly one nonterminal as its superclass, except for the root nonterminal ANY which has no superclass. Each node in the syntax tree will thus be an instance of a subclass to ANY which models the behavior common to all nodes in the tree. The class hierarchy will thus be a tree rooted at ANY, with a top region of abstract nonterminals, lower subtrees of concrete nonterminals, and productions at the leaves.

Abstract nonterminals are similar to the notion of symbol inheritance in [19], which is actually a bit more general since it allows multiple inheritance. This approach could be adopted as an alternative for ORAGs as well.

To be able to refer to each class in the class hierarchy, the productions are named. If a nonterminal *X* has exactly one production, that production will also be named *X*, and both the nonterminal and production are mapped to the same class.

As a generalization of associating attributes with nonterminals and semantic rules with productions, it is possible to also associate attributes with individual productions (local attributes) and semantic rules with nonterminals. A semantic rule in a nonterminal constitutes a default definition that may be overridden by a semantic rule defining the same attribute in a subclass (production or other nonterminal). This notion of overriding is analogous to overriding of virtual functions in object-oriented programming languages.

In order to make sure that the grammar is well-formed, a production or concrete nonterminal $C_1$ that has a concrete nonterminal $C_2$ as a superclass may not declare any inherited attributes. All the inherited attributes of $C_1$ must be declared further up in the class hierarchy, either in an abstract nonterminal or in the topmost concrete nonterminal.

## 5 PicoJava - an example

To illustrate the utility of ORAGs we will demonstrate how name and type analysis can be defined for an object-oriented language. From the point of view of this analysis, our demonstration language PicoJava, a small subset of Java [1], includes the major features of an object-oriented programming language: classes, inheritance, variables, qualified access, and reference assignment. For brevity, methods are omitted but the language allows nested class definitions [22, 26] and global variables, in order to show the combination of block structure and inheritance. The goal of the name analysis is to define a reference attribute decl of each identifier use site, which denotes the corresponding declaration. The goal of the type analysis is to define an attribute tp modelling the type of each expression. We also show how type compatibility for assignments can be specified, in the presence of object-oriented subtyping. The example grammar is non-circular and has been implemented in our language tool APPLAB.

### 5.1 Context-free grammar

Figure 6 shows the context-free grammar of PicoJava in ORAG form. Some remarks about the notation: A nonterminal $X$ appearing to the left of the table cell of another nonterminal or production $C$ is a superclass of $C$. A production $p$: $X_0 \rightarrow X_1 \ldots X_n$ is written "$p$: $\rightarrow X_1 \ldots X_n$" and appears to the right of the table cell for $X_0$. If a nonterminal $X_0$ has only one production, the production takes on the same name as the nonterminal, and is written simply "$\rightarrow X_1 \ldots X_n$". ID is a predefined nonterminal modelling an identifier. The productions for Decls and Stmts make use of a shorthand for lists. The topmost concrete nonterminal, Program, is the start symbol.

### 5.2 Semantic nodes

Several of the nonterminals in the context-free grammar have the prefix SEM. This is a convention for marking so called *semantic nonterminals*, i.e., nonterminals that are not motivated from the context-free syntax point of view, but from an attribution point of view. Semantic nonterminals always have only one production. Thus, by including a semantic nonterminal $S$ on the right hand side of a production $p$, a corresponding $p$-node will get an extra $S$ node as a son, a so called *semantic node*. As an example, the production ClassDecl has a right hand side starting with ID SuperOpt Block, as one would expect, modelling the name of the class, an optional superclass, and a block consisting of declarations and statements. The production continues with two semantic nonterminals: SEMClassStaticEnv SEMClassClassEnv. These latter two nonterminals have only one production each, and a ClassDecl node in the syntax tree will thus always have two extra sons of type SEMClassStaticEnv and SEMClassClassEnv, respectively. Rather than locating all attributes relevant to class declarations directly in ClassDecl, some attributes with a specific purpose can be packaged into a separate semantic nonterminal, e.g. SEMClassStaticEnv. This technique allows an ordinary node to be provided with several interfaces. A reference attribute $r$ can be defined to denote either the ClassDecl node directly, or one of its semantic nodes, depending on what part of the information is relevant to the clients of $r$. This technique is somewhat similar to the use of *part objects* in object-oriented programming [23], where parts of the behavior of an object are delegated to a separate object, that nevertheless forms an integral part of the original object.

**Constant semantic nodes**
When reference attributes are used, it may be the case that an appropriate "real" node cannot be found in the

| *Abstract nonterminals* | | *Concrete nonterminals* | | *Productions* |
|---|---|---|---|---|
| ANY | | Program | | → Block SEMGlobalConstants SEMProgramStaticEnv |
| | | Block | | → Decls Stmts |
| | | Decls | | → Decl* |
| | | Stmts | | → Stmt* |
| | | SEMGlobalConstants | | → SEMEmptyEnv SEMUnknownType |
| | SEMEnv | SEMEmptyEnv | | → |
| | | SEMProgramStaticEnv | | → |
| | | SEMClassStaticEnv | | → |
| | | SEMClassClassEnv | | → |
| | SEMType | SEMUnknownType | | → |
| | | DeclType | | RefDeclType: → UnQualUse |
| | | | | IntDeclType: → |
| | SEMDecl | SEMMissingDecl | | → |
| | | Decl | | ClassDecl: → 'class' ID SuperOpt '{' Block '}' SEMClassStaticEnv SEMClassClassEnv |
| | | | | VarDecl: → DeclType ID |
| | | Stmt | | AssignStmt: → Use '=' Exp |
| | | | | WhileStmt: → 'while' Exp 'do' Stmt |
| | | Exp | Use | UnQualUse: → ID |
| | | | | QualUse: → Use '.' UnQualUse |
| | | SuperOpt | | Super: → 'extends' Use |
| | | | | NoSuper: → |

**Fig. 6.** Context-free syntax for PicoJava

syntax tree. For instance, suppose there is a use of an identifier *x* in a PicoJava program, but no corresponding declaration. In this case, there is no Decl node that the decl attribute of the use site can denote. One solution could be to give the decl attribute the special value null, denoting no node. However, it is often a nicer design to avoid null and instead make use of constant "null objects" [28]. In this case, we introduce a constant node SEMMissingDecl, modelling a missing declaration. This allows clients of the decl attribute to, e.g., access the type of the decl, regardless of if there is a real declaration or not. The type of a missing declaration can be modelled by another "null object", the constant node SEMUnknownType, modelling that the type of the identifier is unknown. An abstract nonterminal SEMDecl is introduced as a common superclass to Decl and SEMMissingDecl in order to be used as the type for the decl attribute. The same pattern is used for SEMUnknownType, where SEMType is introduced as a common superclass of DeclType and SEMUnknownType.

**Global access to constant nodes**

In many cases, it is useful to make the constant nodes globally accessible, i.e., throughout the syntax tree. This is accomplished by collecting all constant nodes under a semantic nonterminal SEMGlobalConstants which is made a semantic node under the start symbol Program. A reference to the SEMGlobalConstants node is

| *Non-terminal* | *Attributes* | *Semantic rules* |
|---|---|---|
| ANY | ↓globals: SEMGlobalConstants | ANY*.globals = globals |
| Program | | ANY*.globals = SEMGlobalConstants |

**Fig. 7.** Specification of the propagation of a reference to global constants

propagated down throughout the syntax tree, thus giving access to all the constant nodes. Figure 7 shows how this can be done conveniently by defining a default semantic rule in the abstract nonterminal ANY which is overridden in Program. The semantic rule in ANY propagates the value of its inherited globals attribute down to all its son nodes of type ANY. Since this holds for all nodes (except for the root Program node which overrides the rule), the reference is propagated down throughout the syntax tree. The overriding rule in Program instead defines globals of its son nodes as denoting the SEMGlobalConstants son node of the Program node. Note that we permit inherited attributes of the start symbol as long as they are not accessed. In this case, Program has an inherited attribute globals since it is a subclass of ANY, but this attribute is never accessed for Program nodes since Program overrides the rule in ANY.

Remarks about the notation. In Figure 7, the sub/superclass relationships between nonterminals and productions are not shown. Please refer to Figure 6 for these relationships. In semantic rules, an attribute *a* of the left hand side nonterminal (or the production) is written without any qualifying name, i.e. simply "*a*", whereas an attribute *b* of a nonterminal *X* of the right-hand side is written "*X.b*". A semantic rule *X*.b = exp* means that the *b* attribute of each right-hand side nonterminal of type *X* is defined to have the value *exp*. The keyword **ref** that we used for RAGs is left out here. Any attribute declared with a nonterminal type is assumed to be a reference.

### 5.3 Modularization

In PicoJava, name and type analysis are dependent on each other. For example, in order to do find the type of a use site, we first need to know its declaration, and in order to find the declaration of a qualified use site, we need to first know the type of the qualifying use site. In order to modularize the definition of this attribution, we first define an interface module consisting of the attributes declared in Figure 8. The Decl.name attribute is simply the name of a Decl node, and the definition of this attribute is so simple that it is given directly in the figure. The definitions of the other three attributes are a bit more complex and are therefore given in a separate modules, making use of the attributes in the interface module. The Exp.tp attribute is a reference to the SEMType node modelling the type for the expression. For expressions where the type is unknown, e.g. uses of undeclared names, the constant node SEMUnknownType is used. The Use.decl attribute is a reference to a SEMDecl node. For declared names, this will be the corresponding Decl node, and for undeclared or multiple declared names it will be the constant node SEMMissingDecl. The ClassDecl.isCircular attribute is a boolean attribute which is *true* if the ClassDecl is part of a circularly defined class hierarchy (which is illegal in PicoJava, but cannot be ruled out by the context-free syntax), and *false* otherwise (the normal case). In the following sections, these attributes are defined.

### 5.4 Name analysis

The goal of the *name analysis* module is to define the Use.decl attribute. The key idea for doing this is to define data structures, constituting of syntax tree nodes and reference attributes, to support the scope rules of PicoJava. For each block-like construct in the language, an attribute decldict containing a dictionary of references to the Decl nodes for local declarations is defined, excluding references to multiply declared identifiers. The blocks are connected to each other so that the declaration of an identifier can be located by doing lookups

| *Nonterminals/productions* | *Attributes and Semantic Rules* |
|---|---|
| Decl | ↑name: **string** |
| ClassDecl | name = ID.val |
| VarDecl | name = ID.val |
| Exp | ↑tp: SEMType |
| Use | ↑decl: SEMDecl |
| ClassDecl | ↑isCircular: **boolean** |

**Fig. 8.** Module declaring name, tp, decl, and isCircular.

| *Nonterminals/ Productions* | *Attributes and Semantic Rules* |
|---|---|
| ANY | ↓env: SEMEnv |
| SEMEnv | SEMDecl **func** lookup(str: **string**) |
| Decls | ↑decldict: **dictionary** (string → Decl) =<br>{(d.name → d) \| d ∈ Decl* ∧ (d.name ∉ {d'.name \| d'∈ Decl* - {d}}) } |
| Block | SEMDecl **func** lookup(str: **string**) =<br>**inspect** $D := Decls.decldict(str)<br>**when** Decl **do** $D<br>**otherwise** globals.SEMMissingDecl |

**Fig. 9.** Module declaring env and SEMEnv.lookup

in block dictionaries in an appropriate order. For Algol-like block structure, a block is connected by a reference attribute to its outer block. For object-oriented inheritance, a class is connected by a reference attribute to its superclass. Semantic nodes that are subclasses of the abstract nonterminal SEMEnv encapsulate these connections and define the function attribute lookup for finding a Decl node for a given identifier. For each node *n* in the syntax tree, an attribute env is defined which refers to a SEMEnv node that connects to the visible identifiers at the point of *n*. The declaration for a Use can be found by calling the lookup function in Use.env. The attribute env thus represents the environment of visible identifiers, similar to the common solution used in canonical attribute grammars, but here env is a reference to a node, possibly connecting to other nodes, rather than a large aggregate attribute.

Figure 9 shows the declaration of ANY.env, the lookup function of SEMEnv, and the definition of decldict. Actually, decldict is an attribute of the Decls node, but is accessed via the function lookup in Block which returns the constant node SEMMissingDecl in case no declaration was found in decldict.

Remarks about the notation. The definition of Block.lookup makes use of an inspect-expression "**inspect** $*V* := *exp* ...", which is similar to a let-expression, but in addition performs a type case. Within each case "**when** *T* **do** *exp*" the named value *V* is guaranteed to have the type *T*. A catch-all clause "**otherwise** *exp*" is needed to make sure there is always an applicable case.

Figure 10 shows the definition of the SEMEnv connections and the SEMEnv.lookup function. There are two block constructs in PicoJava: Program containing global declarations, and ClassDecl, containing declarations local to a class. Algol-like block structure is obtained by nesting a class inside another class. Program has a single semantic node SEMProgramStaticEnv connecting to the Block of the Program (blk). ClassDecl has two semantic nodes; SEMClassClassEnv handles inheritance by connecting to Block of the

| *Nonterminals/Productions* | *Attributes and Semantic Rules* |
|---|---|
| SEMEmptyEnv | lookup(str: **string**) = globals.SEMMissingDecl |
| SEMProgramStaticEnv | ↑ blk: Block = **parent** Program.Block |
| | lookup(str: **string**) = blk.lookup(str) |
| SEMClassClassEnv | ↑ blk: Block = **parent** ClassDecl.Block |
| | ↑ superE: SEMEnv =<br>**if parent** ClassDecl.isCircular<br>**then** globals.SEMEmptyEnv<br>**else parent** ClassDecl.SuperOpt.classE |
| | lookup(str: **string**) =<br>**inspect** $D := blk.lookup(str)<br>**when** Decl **do** $D<br>**otherwise** superE.lookup(str) |
| SEMClassStaticEnv | ↑ thisE: SEMEnv = **parent** ClassDecl.SEMClassClassEnv |
| | ↑ outerE: SEMEnv = env |
| | lookup(str: **string**) =<br>**inspect** $D := thisE.lookup(str)<br>**when** Decl **do** $D<br>**otherwise** outerE.lookup(str) |
| SuperOpt | ↑ classE: SEMEnv |
| Super | classE =<br>**inspect** $D := UnQualUse.decl<br>**when** ClassDecl **do** $D.SEMClassClassEnv<br>**otherwise** globals.SEMEmptyEnv |
| NoSuper | classE = globals.SEMEmptyEnv |

**Fig. 10.** Module defining SEMEnv.lookup

class (blk) and to the SEMClassClassEnv of the superclass (superE); and SEMClassStaticEnv combines inheritance with Algol-like block structure by connecting to the SEMClassClassEnv of the class (thisE) and to the environment (outerE). Figure 11 shows these connections for an example PicoJava program. The lookup function in SEMClassClassEnv is defined to give preference to local declarations over those in the superclass (a declaration in the class will shadow declarations of the same name in superclasses). The lookup function in SEMClassStaticEnv is defined to give preference to inheritance over block structure (a declaration in a superclass will shadow declarations of the same name in an outer block).

Remarks about the notation. The expression "**parent** *T*" is a reference denoting the parent node which must be of type *T*. This is a shorthand for using an inherited attribute parent defined by the parent node. To assure that this expression is always well defined, it is only applicable for nonterminals that appear on the right-hand side of exactly one production.

If the PicoJava program contains an (illegal) circular class structure, care must be taken so that the recursively defined lookup function does not lead to endless recursion. To prevent this, a test on the isCircular attribute (declared in the interface module) is performed when defining the connections between the SEMClassClassEnv nodes. In case the class hierarchy is cyclic, the attribute superE is defined as a reference to the constant node SEMEmptyEnv rather than to the SEMClassClassEnv of the superclass. This way, the graph consisting of SEMClassClassEnv nodes and superE attributes can never be cyclic, and their lookup functions will therefore terminate.

Figure 12 shows the definition of env. For most nodes, the environment is the same as for the enclosing node, as defined by the default semantic rule in ANY. This default behavior is overridden in three productions. In Program and ClassDecl, the environment for the Block is defined as a reference to the SEMProgram-

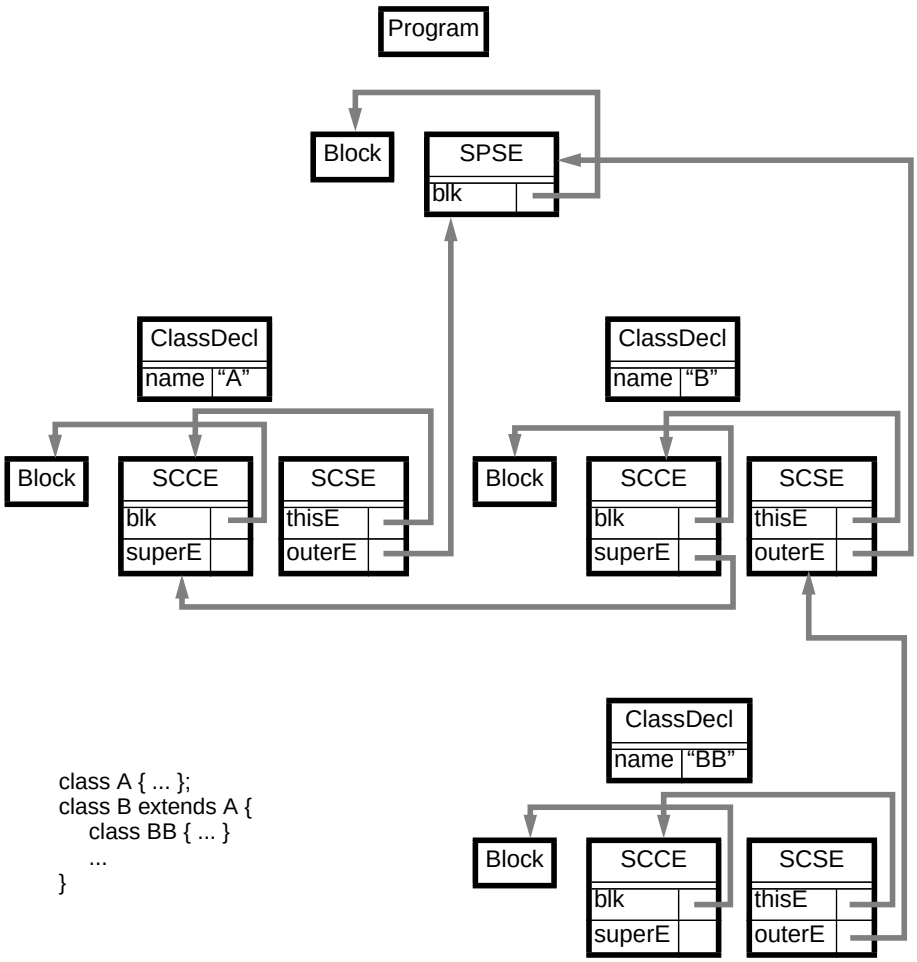


**Fig. 11.** Connections between SEMEnv nodes for a small program

| *Nonterminals/Productions* | *Attributes and Semantic Rules* |
|---|---|
| ANY | ANY*.env = env |
| Program | Block.env = SEMProgramStaticEnv |
| ClassDecl | Block.env = SEMClassStaticEnv |
| QualUse | UnQualUse.env =<br>**inspect** $T := Use.tp<br>**when** RefDeclType **do**<br>**inspect** $D := $T.UnQualUse.decl<br>**when** ClassDecl **do** $D.SEMClassClassEnv<br>**otherwise** globals.SEMEmptyEnv<br>**otherwise** globals.SEMEmptyEnv |

**Fig. 12.** Module defining env

| *Nonterminals/Productions* | *Attributes and Semantic Rules* |
|---|---|
| UnQualUse | decl = env.lookup(ID.val) |
| QualUse | decl = UnQualUse.decl |

**Fig. 13.** Module defining decl

StaticEnv and SEMClassStaticEnv, respectively. In the QualUse production, the environment of the second operand depends on the type of the first operand which should be a reference variable.

The definition of the decl attribute is now simple, as shown in Figure 13.

### 5.5 Check of circular class hierarchy

Figure 14 shows the definition of the isCircular attribute declared in Figure 8 which says if a class is circularly defined or not. The idea is to use a help function circularClass(s) which is called recursively for each ClassDecl in the superclass chain. The argument s contains the set of references to already visited ClassDecl nodes. The recursion is terminated either when the top of the class hierarchy is reached (the normal case), or when a ClassNode is reached that is already in s (a cycle is found in the hierarchy).

| *Nonterminals/Productions* | *Attributes and Semantic Rules* |
|---|---|
| ClassDecl | isCircular = SuperOpt.circularClass({**self**}) |
| | **boolean func** circularClass (s: **set of** ClassDecl) =<br>**if self** ∈ s<br>**then true**<br>**else** SuperOpt.CircularClass(s ∪ {**self**}) |
| SuperOpt | **boolean func** circularClass (s: **set of** ClassDecl) |
| NoSuper | circularClass(s: **set of** ClassDecl) = **false** |
| Super | circularClass(s: **set of** ClassDecl) =<br>**inspect** $D := UnQualUse.Decl<br>**when** ClassDecl **do** $D.circularClass(s)<br>**otherwise false** |

**Fig. 14.** Module defining isCircular

Remark on the notation. The construct "**self**" in a rule means a reference to the left-hand nonterminal of the production. E.g., in Figure 14, **self** refers to the ClassDecl node.

### 5.6 Type analysis

Figure 15 shows the definition of the tp attribute declared in Figure 8. For illegal uses of identifiers, e.g. where the declaration is missing, the constant node SEMUnknownType is used.

The tp attribute can be used to perform type checking, e.g., checking that the types of the left and right hand side of an assignment are compatible. For an object-oriented language, this check is rather more involved than for procedural languages, due to the subtype compatibility rules. For a reference assignment *Use* = *Exp* in PicoJava, the class of *Exp* must be the same or a subclass of the class of *Use*. To further show the expressiveness of RAGs, Figure 16 shows how a boolean attribute typesCompatible can be defined for Assignment, taking into account both ordinary types and reference types with subtyping. The typesCompatible

attribute is *true* if the assignment statement is type correct. A help function assignableTo is defined in SEMType such that T1.assignableTo(T2) is *true* if it is legal to assign a value of type T1 to a variable of type T2. For reference types (RefDeclType), this function checks if the class of T1 is a subclass of that of T2. To perform this check, the class hierarchy is traversed using a recursive function recSubclassOf in ClassDecl.

| *Nonterminals/Productions* | *Attributes and Semantic Rules* |
|---|---|
| Use | tp =<br>**inspect** $D := decl<br>**when** VarDecl **do** $D.DeclType<br>**otherwise** globals.SEMUnknownType |
| QualUse | tp = UnQualUse.tp |

**Fig. 15.** Module defining tp

| *Nonterminals/Productions* | *Attributes and Semantic Rules* |
|---|---|
| SEMType | **boolean func** assignableTo(T: SEMType) |
| SEMUnknownType | assignableTo(T: SEMType) = **false** |
| IntDeclType | assignableTo(T: SEMType) = T **in** IntDeclType |
| RefDeclType | assignableTo(T: SEMType) =<br>**inspect** $T := T<br>**when** RefDeclType **do**<br>**inspect** $D := UnQualUse.decl<br>**when** ClassDecl **do**<br>**inspect** $DT := $T.UnQualUse.decl<br>**when** ClassDecl **do** $D.subclassOf($DT)<br>**otherwise false**<br>**otherwise false**<br>**otherwise false** |
| ClassDecl | **boolean func** subclassOf(C: ClassDecl) =<br>**if** isCircular<br>**then false**<br>**else** recSubclassOf(C) |
| | **boolean func** recSubclassOf(C: ClassDecl) =<br>**if** C = **self**<br>**then true**<br>**else**<br>**inspect** $Super := SuperOpt.superClass<br>**when** ClassDecl **do** $Super.recSubclassOf(C)<br>**otherwise false** |
| AssignStmt | ↑typesCompatible: **boolean** = Exp.tp.assignableTo(Use.tp) |
| SuperOpt | ↑superClass: ClassDecl |
| NoSuper | superClass = **null** |
| Super | superClass =<br>**inspect** $D := SimpleUse.decl<br>**when** ClassDecl **do** $D<br>**otherwise null** |

**Fig. 16.** Module defining Assignment.typesCompatible

However, in order to make sure that this function terminates, even in the case of an illegal circular class hierarchy, the attribute isCircular is checked before calling the recursive function (in ClassDecl.subclassOf).

## 6 Experimental system

We have implemented RAGs in our language tool APPLAB and used RAGs to specify a number of languages, including an extended version of PicoJava described in Section 5 (the extended version includes also methods and some additional basic types, operators, and statements). We are also working with specification of worst-case execution time analysis [24], specifications of robot languages [6], state transition languages, and the RAG formalism itself.

The APPLAB system is an interactive language tool where both programs and grammars for the programming language can be edited at the same time, resulting in a highly flexible and interactive environment for language design. Changes to the grammars, e.g. changes to the context-free syntax or changes to the attributes and semantic functions, are immediately reflected in the language-based program editor, allowing the user to get immediate feedback on the effects of changes to the grammar specification. The details of APPLAB are covered in [5, 6] (although these papers do not focus on RAGs which is a later addition).

Figure 17 shows a screendump from the APPLAB system, showing the editing of an example program in PicoJava, and parts of the grammar specification. The two windows labelled typesCompatible show the val-

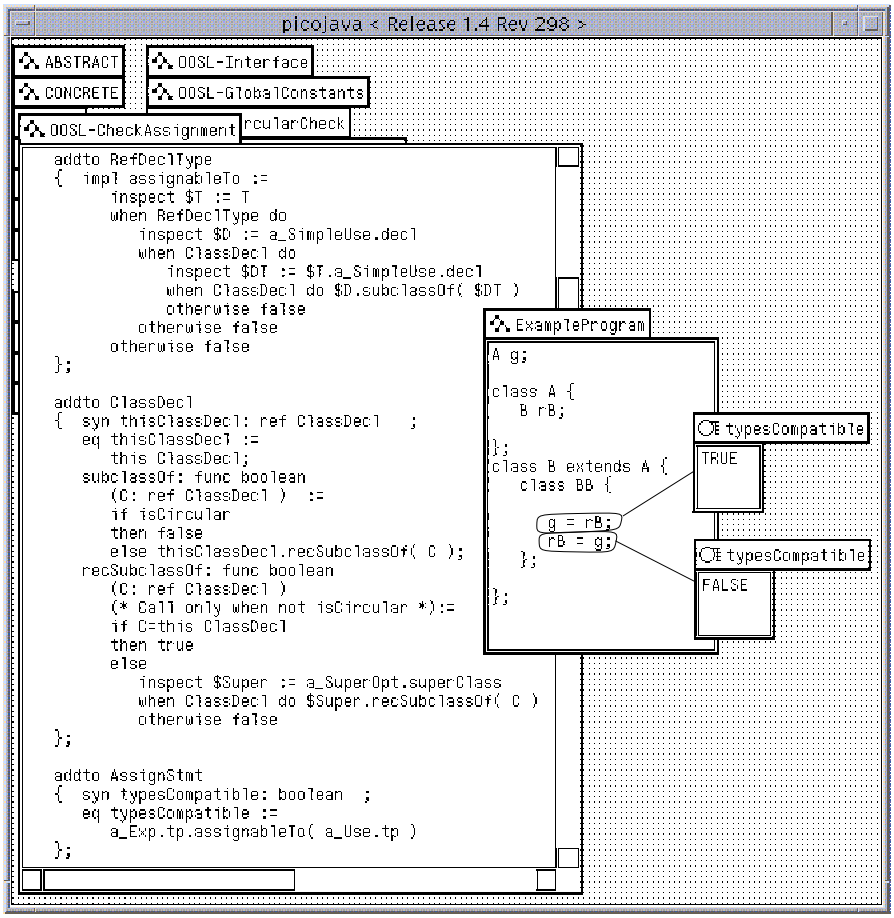


**Fig. 17.** Screendump from APPLAB. The typesCompatible windows show the corresponding attribute values for the two assignment statements in class BB.

ues of the corresponding attribute for the two assignment statements in class BB which is an inner class of B which is a subclass of A. The example illustrates both block structure (g is declared globally, i.e. two levels outside of BB) and combined block structure and inheritance (rB is declared one level outside of BB in a superclass of B). The first assignment is correct (typesCompatible=TRUE) since the class of rB (B) is a subclass of the class of g (A). The second assignment is not correct (typesCompatible=FALSE) since A is not equal to or a subclass of B.

## 7 Related work

The idea to support non-local dependencies has been suggested in a number of systems in various ways. Many of these approaches provide some kind of special support for nested scopes (supporting Algol-like block structure) such as [13, 14, 3, 17, 11, 2], but fail to handle more complex scope combinations such as inheritance or qualified access of identifiers. In contrast, RAGs allow the scope mechanisms to be defined completely within the grammar formalism and are not restricted to predefined combinations.

Some systems support limited kinds of reference attributes, allowing the syntax tree to be extended to acyclic or even cyclic graphs, but where reference attributes are not allowed to be dereferenced in order to define other attributes. Thus, they cannot be used for specifying name analysis in the concise way we have shown above, where, e.g., inheritance chains are traversed during name lookup and type checking. In particular:

- The Synthesizer Generator supports *syntactic references*, i.e., an attribute may be a reference to a syntax tree node [25]. However, attributes of the referenced node may not be accessed via the reference attribute. I.e., the syntactic references are considered to stand for *unattributed* subtrees. There are certain similarities to RAGs in that the syntax tree can itself be used as e.g. symbol tables, rather than having to construct such information in a separate attribute domain. However, RAG reference attributes are much more powerful than syntactic references in that the attributes of the referenced nodes may be accessed, allowing attribute information to be propagated along non-locally paths.
- The Synthesizer Generator also allows attributes to be defined as references to other attributes. The is used to define cyclic graphs in code generation, e.g. for linking the last instruction of a while statement back to the first instruction. However, for the purpose of the attribute evaluation, these references are just treated as constants and may not be dereferenced. Dereferencing can only be done after the attribution is complete, by an interpreter written directly in C.
- The Elegant system [2] also supports the construction of a cyclic *program construct graph* which is essentially the syntax tree extended with edges from use sites to declaration sites. However, the additional edges cannot be dereferenced in order to define other attributes. They may, however, be dereferenced after the attribution is complete, in order to check context conditions. The resulting program construct graph can also be processed by a special-purpose code generation formalism.

There is some earlier work which aims at fuller support for reference attributes.

- In our previous work on *Door Attribute Grammars* [9, 10] dereferencing of reference attributes is supported, but may only take place in special nonterminals called *doors*. This way, the non-local dependencies are encapsulated in a so called *door package*. Door AGs also support remote definition where collection values can be defined remotely via references. Door AGs support efficient incremental attribute evaluation, but the implementation is not fully automatic because the door package needs to be implemented manually. Door AGs allows object-oriented languages to be specified in a way very similar to for RAGs, using similar techniques for connecting environments and traversing inheritance graphs, but RAGs are considerably more compact because the non-locally accessed information does not need to be propagated to door nonterminals, but can be accessed directly, thus avoiding replication of information. RAGs offer fully automatic evaluation, but not (currently) incremental attribute evaluation.
- Vorthmann has developed a graphical technique called *visibility networks* for describing name analysis and use-declaration bindings in programming languages, and exemplified the technique for Ada [27]. Also

here, the focus is on providing efficient incremental evaluation. This technique might be interesting to integrate with RAGs in order to provide support for incremental attribute evaluation for certain classes of RAGs.

- Boyland has addressed the problem of computing static evaluation schemes for grammars with both remote access and remote definition via reference attributes in order to apply visit-oriented evaluation algorithms. However, the scope of his technique is unclear. It has been applied only for simple example grammars and does not seem to be implemented [4].

## 8 Conclusions

We have presented Reference Attributed Grammars (RAGs) and showed how they can be applied to an advanced problem: name and type analysis for an object-oriented language, yielding a simple and concise non-circular specification. We have implemented the RAG formalism and an evaluation algorithm that can handle any non-circular RAG. In our tool for language experimentation, APPLAB, it is possible to experiment with RAG specifications and immediately try out changes to the attribution rules, e.g. by asking for the values of attributes in an example program.

There are several advantages of RAGs over canonical AGs. First, there is no need in RAGs to replicate the information available in the syntax tree into attributes. By using reference attributes the syntax tree itself can be used as the information source. The syntax nodes can be connected using reference attributes to form suitable data structures, also cyclic ones, without the need for introducing data structures and functions in auxiliary languages. Second, the semantic functions working on a complex data structure can be split into smaller functions, delegated to the different syntax nodes making up the data structure, and specified completely within the RAG formalism. Third, it is easy to extend an existing grammar with additional functionality. This was shown in the PicoJava example where the test for type compatibility of assignments was added in a very concise way, although it included advanced rules for subtype compatibility.

In our experience, RAGs are of immediate practical use and we have a number of current projects concerning language specification using this technique. There are many interesting areas for further research, including the following.

- Efficient incremental evaluation of RAGs is an open problem. However, RAGs are a much better starting point for incremental evaluation than canonical AGs since large aggregate attributes are not needed in RAGs, and the number of affected attributes after a change is much lower than for a canonical AG.
- It would be useful to develop algorithms for deciding statically if a RAG is non-circular. This is an open problem. The APPLAB system currently reports circular dependencies at evaluation time.
- It would be useful to develop algorithms for deciding if a RAG contains nonterminating semantic functions. In the PicoJava example there are two cases where special care is taken in order to make sure that the semantic functions terminate, namely when using recursive functions that traverse the class hierarchy. The attribute isCircular was introduced in order to be able to terminate the recursion in case of a cyclic class hierarchy. During grammar development it would be useful if potential circular structures and nonterminating functions could be automatically spotted by the system.
- The formalism should be extended so that semantic nonterminals and nodes can be added in extension modules, i.e. without having to modify the context-free syntax.
- Since RAGs allow arbitrary data structures to be built using syntax tree nodes and references it should be interesting to extend the technique to allow graph-based grammars, working on syntax graphs rather than trees. This would be relevant for building language-based editors for, e.g., UML class diagrams or state-transition diagrams.

## Acknowledgements

This work was supported by NUTEK, the Swedish National Board for Industrial and Technical Development. Elizabeth Bjarnason implemented the major parts of the APPLAB system, making it easy to add the support for RAGs.

## References

1. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley. 1996.
2. L. Augusteijn. The Elegant Compiler Generator System. In *Attribute Grammars and their Applications*, pp 238-254, LNCS 461, Springer-Verlag, September 1990.
3. G. M. Beshers and R. H. Campbell. Maintained and constructor attributes. In *Proceedings of the SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 34–42, Seattle, Wa., 1985. ACM. SIGPLAN Notices, 20(7).
4. John 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.
5. E. Bjarnason. *Interactive Tool Support for Domain-Specific Languages*. Licentiate thesis. Dept. of Computer Science, Lund University, December 1997.
6. E. Bjarnason, G. Hedin, K. Nilsson. Interactive Language Development for Embedded Systems. To appear in *Nordic Journal of Computing*. 1998.
7. 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*, 85–98, July 1986. ACM. SIGPLAN Notices, 21(7).
8. G. Hedin. An object-oriented notation for attribute grammars. *ECOOP'89*. BCS Workshop Series, pp 329-345, Cambridge University Press. 1989.
9. G. Hedin. *Incremental Semantic Analysis*. PhD thesis, Dept. of Computer Science, Lund University, Sweden, March 1992.
10. G. Hedin. An overview of door attribute grammars. *International Conference on Compiler Construction (CC'94)*. LNCS 786, Springer Verlag. 1994.
11. R. Hoover. *Incremental Graph Evaluation*. PhD thesis, Cornell University, Ithaca, N.Y., May 1987. Tech. Rep. 87-836.
12. F. Jalili. A general linear time evaluator for attribute grammars. *ACM SIGPLAN Notices, Vol 18(9):35-44, Sept. 1983*.
13. G. F. Johnson and C. N. Fischer. Non-syntactic attribute flow in language based editors. In *Proc. 9th POPL*, pp 185–195, Albuquerque, N.M., January 1982. ACM.
14. G. F. Johnson and C. N. Fischer. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In *Proc. 12th POPL*, pages 141–151, New Orleans, La., January 1985. ACM.
15. L. G. Jones. Efficient evaluation of circular attribute grammars. *ACM TOPLAS*, 12(3):429–462, 1990.
16. M. Jourdan. An optimal-time recursive evaluator for attribute grammars. In M. Paul and B. Robinet, editors, *International Symposium on Programming, 6th Colloquium*, pp 167–178, LNCS 167. Springer-Verlag, 1984.
17. G. Kaiser. *Semantics for Structure Editing Environments*. PhD thesis, Carnegie-Mellon University, Pittsburgh, Pa., May 1985. CMU-CS-85-131.
18. U. Kastens. Ordered attributed grammars. *Acta Informatica*, 13:229–256, 1980.
19. U. Kastens and W. M. Waite. Modularity and Reusability in Attribute Grammars. *Acta Informatica*, 31:601-627, 1994.
20. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127-145, June 1968.
21. 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.
22. O. L. Madsen. Block structure and object-oriented languages. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. The MIT Press, 1987.
23. O. L. Madsen and 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.
24. P. Persson and G. Hedin. Interactive Execution Time Predictions Using Reference Attributed Grammars. To appear at *WAGA'99*. 1998.
25. T. W. Reps and T. Teitelbaum. *The Synthesizer Generator. A system for constructing language-based editors*. Springer Verlag. 1989.
26. Sun Microsystems. *Inner Classes Specification*. 1996. http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses
27. S. A. Vorthmann. *Modelling and Specifying Name Visibility and Binding Semantics*. CMU-CS-93-158. Carnegie Mellon University, Pittsburgh, Pa., July 1993.
28. B. Woolf. Null Object. In R. Martin et al. (eds), *Pattern Languages of Program Design* 3, pp 5-18, Addison Wesley, 1998.