

Reference Attributed Grammars

Görel Hedin
Dept. of Computer Science
Lund University
Sweden
E-mail: Gorel.Hedin@cs.lth.se

Keywords: Attribute Grammars, Object-Oriented Languages, Reference Attributes, Remote Attribute Access

Edited by:

Received:

Revised:

Accepted:

An object-oriented extension to canonical attribute grammars is described, 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 in a concise and modular manner in these grammars, and an optimal evaluation algorithm is available. An extensive example is given, capturing all the key constructs in object-oriented languages including block structure, classes, inheritance, qualified use, and assignment compatibility in the presence of subtyping. The formalism and algorithm have been implemented in APPLAB, an interactive language development tool.

1 Introduction

Canonical attribute grammars (AGs), as introduced by Knuth [26], 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, 5, 12, 14, 15, 16, 17, 19, 20, 23, 33, 34, 38]. Our approach is in the line of our earlier work [12, 14, 15, 16], of Poetzsch-Heffter [33, 34], and of Boyland [4] in that we propose an extension that permits attributes to be explicit references denoting nodes arbitrarily far away in the syntax tree, and attributes of those nodes to be accessed via such reference attributes. Similar to Poetzsch-Heffter and Boyland we propose a recursive evaluation algorithm that allows optimal evaluation for non-circular AGs with such extensions. The formalism we propose, Reference Attributed Grammars (RAGs), casts these extensions into an object-oriented form, allowing advanced static-semantic analysis prob-

lems to be expressed in a concise and modular manner. We give an extensive example of this by providing a complete specification of PicoJava, a small subset of Java including key constructs found in object-oriented languages such as block structure, classes, inheritance, qualified use, and assignment compatibility in the presence of subtyping. We have implemented the formalism and evaluation algorithm in our interactive language development tool APPLAB (APPLication language LABoratory) [6, 7].

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 the basic RAG formalism, discusses the evaluation algorithm, and compares RAGs to canonical AGs. Section 4 discusses additional object-oriented features of RAGs, including a class hierarchy for nonterminals 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 nonterminals and semantic rules for the productions. The attributes are characterized as *synthesized* or *inherited*, depending on if they are used to transmit in-

formation upwards or downwards in the syntax tree. Given a production $X_0 \rightarrow X_1 \dots X_n$, a semantic rule is written $a_0 = f(a_1, \dots, a_m)$ and defines a_0 as the value of applying the *semantic function* f to the attributes a_1, \dots, a_m . The attribute a_0 must be either a synthesized attribute of X_0 or an inherited attribute of $X_j, 1 \leq j \leq n$. 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, $a_k, 1 \leq k \leq m$, must be an attribute of $X_j, 0 \leq j \leq n$. 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 $X_0 \rightarrow X_1 \dots X_n$ to have exactly one rule for each synthesized attribute of X_0 and one rule for each inherited attribute of $X_j, 1 \leq j \leq n$.

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 each 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 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. [10, 21].

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 may depend 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 kind, 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 kinds and is defined analogously to the environment mapping names to types. Just like the type environment, the definition of the kind environment needs to involve all intermediate nodes. A second alternative is to modify the original type environment to also include kind 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 illustrate 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

<i>Nonterminal</i>	<i>Attributes</i>	<i>Productions</i>	<i>Semantic rules</i>
A		$A \rightarrow BC$	$B.rC = C$ $C.rB = B$
B	$\downarrow rC: \mathbf{ref}(C)$ $\uparrow b: \mathbf{integer}$	$B \rightarrow$	$B.b = B.rC.c$
C	$\downarrow rB: \mathbf{ref}(B)$ $\uparrow c: \mathbf{integer}$	$C \rightarrow$	$C.c = 7$

Figure 1: RAG specification of TINY

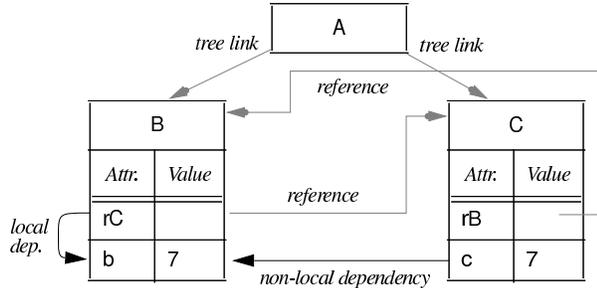


Figure 2: RAG attribution of TINY (non-circular)

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 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 nonterminal X_1 in a production is defined using an attribute of a nonterminal 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 (a, b) occurs when b is defined by a semantic function

that accesses a via a reference attribute r . The dependency (a, b) 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 [24] 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, as also noted by [33] and [4]. By caching an attribute value at the first access and returning the cached value at subsequent accesses, this evaluation algorithm becomes optimal. Setting a flag for attributes under evaluation allows circularities in the grammar to be found at evaluation time. Several implementations of this algorithm have been presented for canonical attribute grammars [27, 18, 22]. In our system (APPLAB), we have implemented the algorithm for RAGs by using techniques from object-oriented programming, as described for canonical AGs in [13]. 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.

3.4.1 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

	Attributes		Semantic rules
A	\uparrow id: integer \uparrow ct: $\langle \rangle$ \uparrow subCt: array[tuple] \uparrow allCt: array[tuple]	$A \rightarrow B C$	$B.rC = C.id$ $C.rB = B.id$ $A.id = 1$ $B.id = A.id + 1$ $C.id = B.maxId + 1$ $A.ct = \langle \rangle$ $A.subCt =$ $[A.id \rightarrow A.ct] \cup$ $B.subCt \cup$ $C.subCt$ $A.allCt = A.subCt$ $B.allCt = A.allCt$ $C.allCt = A.allCt$
B	\downarrow rC: integer \uparrow b: integer \downarrow id: integer \uparrow maxId: integer \uparrow ct: \langle integer, integer \rangle \uparrow subCt: array[tuple] \downarrow allCt: array[tuple]	$B \rightarrow$	$B.b = B.allCt[B.rC](2)$ $B.maxId = B.id$ $B.ct = \langle B.rC, B.b \rangle$ $B.subCt =$ $[B.id \rightarrow B.ct]$
C	\downarrow rB: integer \uparrow c: integer \downarrow id: integer \uparrow maxId: integer \uparrow ct: \langle integer, integer \rangle \uparrow subCt: array[tuple] \downarrow allCt: array[tuple]	$C \rightarrow$	$C.c = 7$ $C.maxId = C.id$ $C.ct = \langle C.rB, C.c \rangle$ $C.subCt =$ $[C.id \rightarrow C.ct]$

Figure 3: Table-translated specification of TINY (canonical AG form)

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 pre-order 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 $\langle a_1, \dots, a_k \rangle$ where a_1, \dots, 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 .

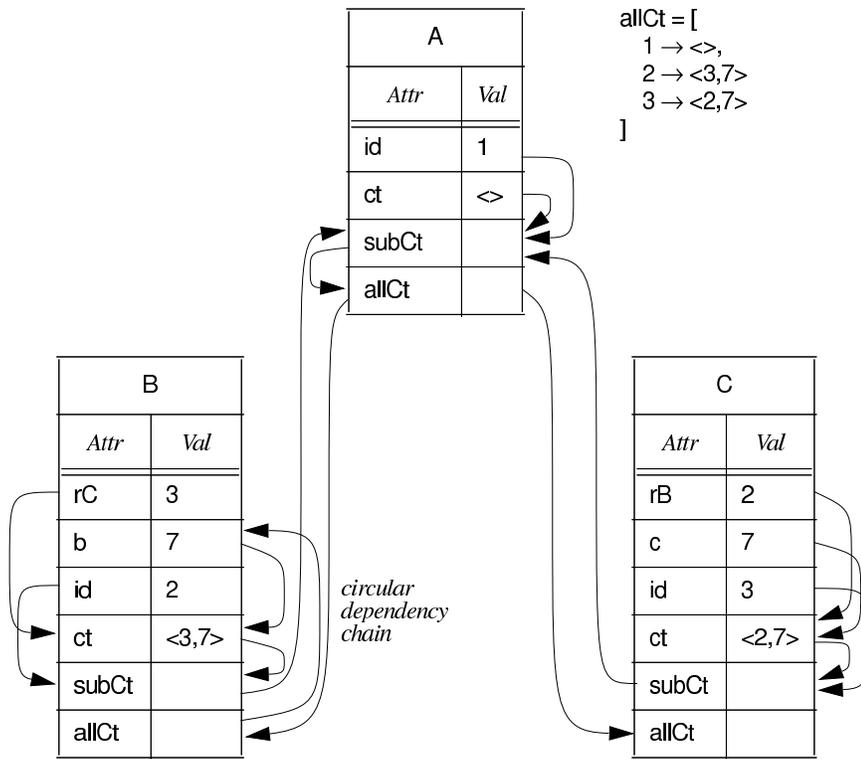


Figure 4: Attribution of TINY for table-translated specification (circular)

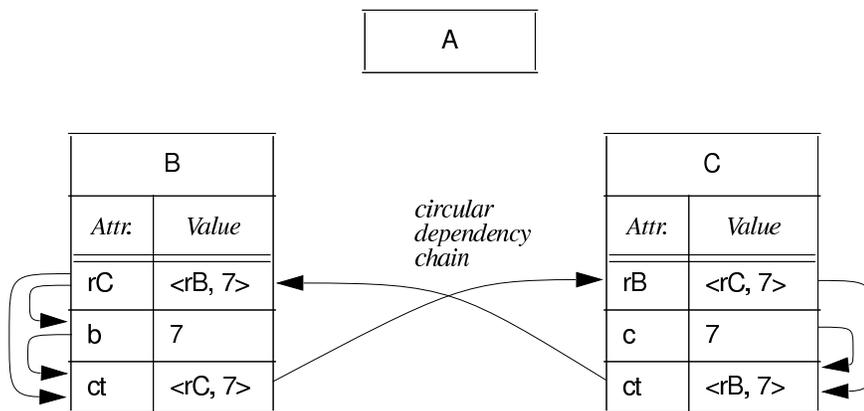


Figure 5: Attribution of TINY for substitution-translated specification (circular). Attributes rC and rB have infinite attribute values

- In semantic rules, an access to a symbol X (used as a reference value) is replaced by the expression $X.\text{id}$, 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 $\text{allCt}[r](i)$, 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.

3.4.2 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 \mathbb{C} with an attribute $\text{d} = \text{rB.b}$.

4 Object-oriented features of RAGs

In this section, we will introduce some features of RAGs which make specifications more concise. These features are based on an object-oriented view of attribute grammars, where nonterminals are viewed as

superclasses and productions as subclasses. In particular, we will discuss the use of virtual function attributes and an extended class hierarchy of nonterminals.

4.1 Virtual function attributes

Canonical AGs have a straight-forward translation to object-oriented programming [13]. 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.

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

It is possible to eliminate virtual function attributes and replace them by auxiliary functions. Each semantic rule defining the attribute is then replaced by an auxiliary function, and type case analysis is used at each call site to call the correct auxiliary function. This translation is analogous to translating object-oriented programs to procedural programs. Thus, virtual function attributes are not strictly necessary. However, they make the grammar more modular and easy to extend and change, by allowing the call site expressions to be written in a polymorphic way (being able to handle objects of different types without

having to mention these types explicitly).

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 [25], but makes use of single rather than multiple inheritance. We have chosen single inheritance because we find it conceptually simpler and because we replace the use of multiple inheritance by composition, using so called semantic nodes as explained in Section 5.2.

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 RAGs 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 [28, 37] 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 RAG 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 $p : X_0 \rightarrow X_1 \dots X_n$ is written " $p \rightarrow X_1 \dots X_n$ " 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 " $\rightarrow X_1 \dots X_n$ ". **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

<i>Abstract nonterminals</i>		<i>Concrete nonterminals</i>	<i>Productions</i>
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: →

Figure 6: Context-free syntax for PicoJava

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 [29], where parts of the behavior of an object are delegated to a separate object, that nevertheless forms an integral part of the original object.

5.2.1 Constant semantic nodes

When reference attributes are used, it may be the case that an appropriate "real" node cannot be found in the 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" [39]. 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

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

Figure 7: Specification of the propagation of a reference to global constants

introduced as a common superclass of DeclType and SEMUnknownType.

5.2.2 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 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 in Section 3.2 is left out here. Any attribute declared with a nonterminal type is assumed to be a reference.

Nonterminals and 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

Figure 8: Module declaring name, tp, decl, and isCircular

5.3 Modularization

In PicoJava, name and type analysis are dependent on each other. For example, in order to 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 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 multiply 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 de-

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

Figure 9: Module declaring env and lookup

clared identifiers. The blocks are connected to each other so that the declaration of an identifier can be located by doing lookups 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 node is connected by a reference attribute to its superclass node. 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 analysis. 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

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

Figure 10: Module defining lookup

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 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

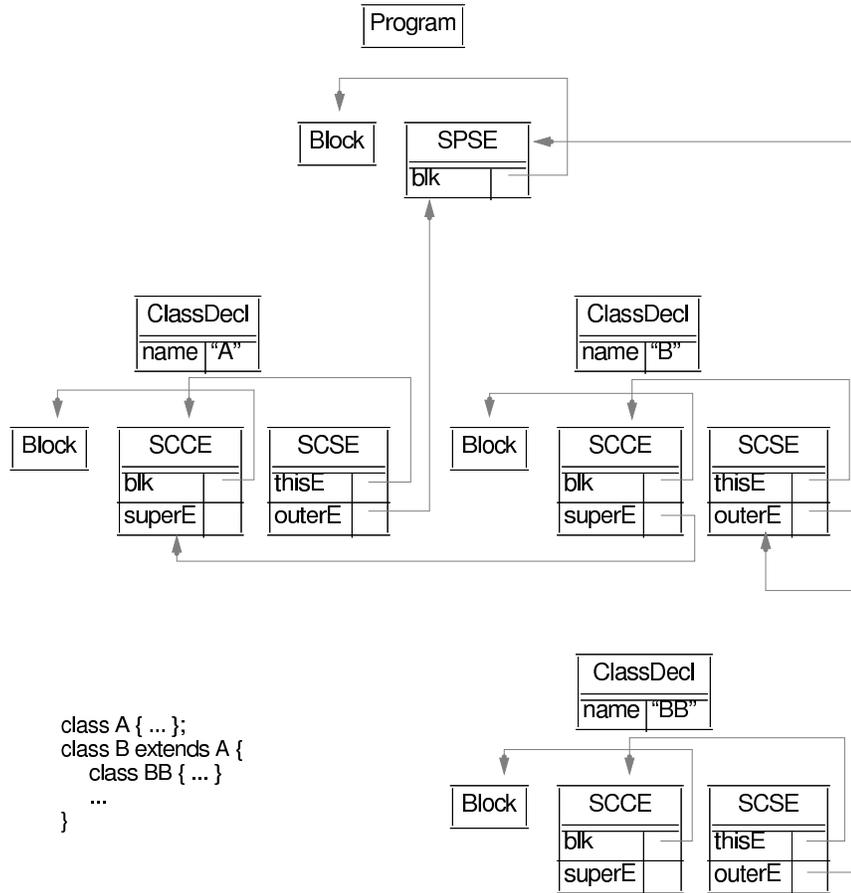


Figure 11: Connections between `SEMEnv` nodes for a small program

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.

A PicoJava program may contain an (illegal) circular class structure. Therefore, 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 pro-

ductions. In `Program` and `ClassDecl`, the environment for the `Block` is defined as a reference to the `SEMProgramStaticEnv` 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).

Remark on the notation. The construct `"self"` in a rule means a reference to the left-hand nonterminal of

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

Figure 12: Module defining env

<i>Nonterminals/ Productions</i>	<i>Attributes and Semantic Rules</i>
UnQualUse	decl = env.lookup(ID.val)
QualUse	decl = UnQualUse.decl

Figure 13: Module defining decl

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 refer-

<i>Nonterminals/ Productions</i>	<i>Attributes and Semantic Rules</i>
	isCircular = SuperOpt.circularClass(self)
ClassDecl	boolean func circularClass (s: set of ClassDecl) = if self ∈ s then true 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) = inspect \$D := UnQualUse.decl when ClassDecl do \$D.circularClass(s) otherwise false

Figure 14: Module defining isCircular

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

Figure 15: Module defining tp

ence 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`. 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 [31, 32], robot languages [7], state transition languages [11], visualization [30], design patterns [8, 9], and the RAG formalism itself.

The APPLAB system is an interactive language tool where both programs and grammars for the program-

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

Figure 16: Module defining typesCompatible

ming 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 [6, 7] (although these papers do not focus on reference attributes 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. In the ExampleProgram window, the user has selected the statement `g=rB` in class `BB`, where `BB` is an inner class of `B` which in turn 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 assignment is type correct (the value of `typesCompatible` is `TRUE`) since `B` (the class of `rB`) is a subclass of `A` (the class of `g`). The value of the attribute is shown in a separate attribute window at the user's request (after selecting the attribute in a popup-menu). The subsequent assignment `rB=g` is not type correct since `A` (the class of `g`) is not equal to or a subclass of `B` (the class of `rB`), and a request for the `typesCompatible` attribute of that statement would display a corresponding attribute window showing that `typesCompatible` has the value `FALSE`.

7 Related work

The idea to support non-local dependencies has been suggested in a number of systems in various ways. Early approaches provided special support for nested scopes (supporting Algol-like block structure) such as [19, 20, 3, 23, 17, 2], but fail to handle more complex scope combinations such as inheritance or qualified access of identifiers. Later approaches support explicit reference attributes and remote attribute access, in a similar way as described here, and allows scope mechanisms to be defined without being restricted to pre-defined combinations. In particular:

- In our previous work on Door Attribute Grammars [14, 15, 16] dereferencing of reference attributes is supported, but must be delegated to 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.
- The MAX system by Poetzsch-Heffter [33, 34] supports reference attributes and remote access, and develops an extension to term algebras called *occurrence algebras* to formalize the approach. A demand-based evaluation technique is used, and in addition an approximate static dependency

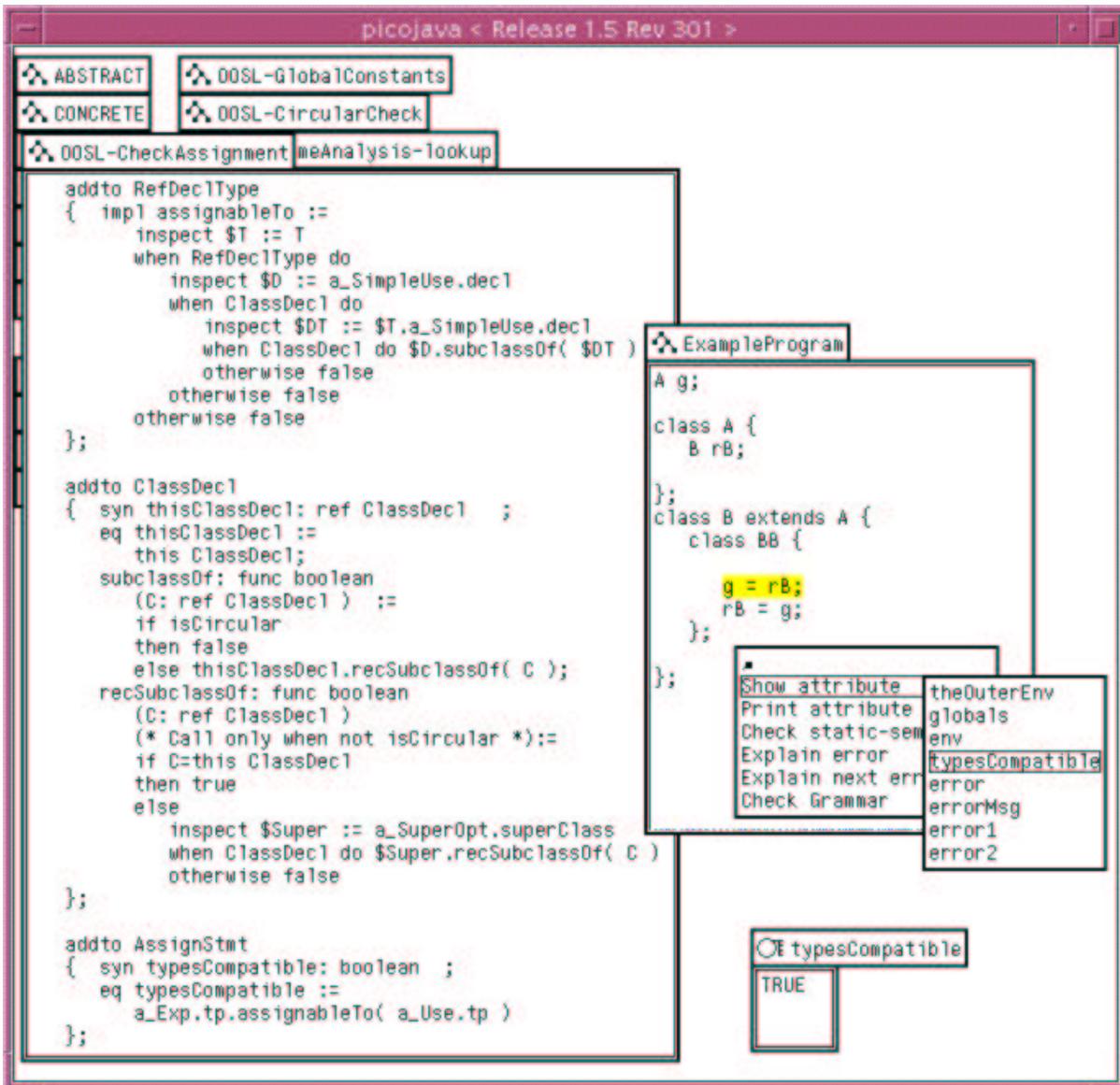


Figure 17: Screenshot from APPLAB. The attribute `typesCompatible` is shown for the current focus in the `ExampleProgram` window (the assignment statement `g = rB` in class `BB`)

analysis is developed which allows many function calls to be eliminated and thereby speed up the evaluation [34].

- Boyland also developed a system supporting both remote access and remote definition, and making use of a demand-algorithm for attribute evaluation [4]. He has also 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 this latter technique is unclear. It has been applied only for simple example grammars and does not seem to be implemented [5].
- Sasaki and Sassa have developed a static evaluation scheme for circular grammars with reference attributes and remote access [36]. Their motivating example is liveness analysis in the presence of `gotos` where the `goto` links are modelled by reference attributes in the AST. In their evaluation scheme remote dependencies are added conservatively, causing cycles in the production dependency graphs that correspond to real or potential cycles in an actual tree. Cycles are evaluated iteratively.

The underlying principles of remote access and attribute evaluation are the same in RAGs as in MAX and in Boylands system. However, the RAG formulation is radically different, expressing the specification using object-oriented concepts like inheritance and virtuals.

Other related approaches include the following:

- The Synthesizer Generator supports syntactic references, i.e., an attribute may be a reference to a syntax tree node [35]. 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. This 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.
- 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 [38]. 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.

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. Figures 6–10 and 12–16 constitute a complete static-semantic specification of PicoJava, a language with all the key object-oriented constructs: block structure, classes (including inner or nested classes), inheritance, qualified use, and assignment compatibility in the presence of subtyping.

The use of reference attributes allows cyclic structures to be constructed on top of the syntax tree substrate. We have demonstrated how attributes can be used to check for such cyclic structures to ensure that semantic functions terminate, thus allowing the RAG to remain non-circular, although it works on cyclic structures. (See the definition and use of the `ClassDecl.isCircular` attribute in Section 5.)

We have implemented the RAG formalism and an evaluation algorithm that can handle any non-circular RAG. In our tool for language experimentation, AP-PLAB, 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.

We have demonstrated 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 tests circularity dynamically and 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. We expect this to be straight-forward, making use of object-oriented concepts like part objects and inner (anonymous) classes as available in BETA and recently also in Java [28, 29, 37].
- 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 reference attributes.

References

- [1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley. 1996.
- [2] L. Augustejn. 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*, pp 34–42, Seattle, Wa., 1985. ACM SIGPLAN Notices, 20(7).
- [4] J. T. Boyland. *Descriptive Composition of Compiler Components*. PhD thesis, Report No. UCB//CSD-96-916, Computer Science Division (EECS), University of California, Berkeley, California, September 1996.
- [5] J. 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.
- [6] E. Bjarnason. *Interactive Tool Support for Domain-Specific Languages*. Licentiate thesis. Dept. of Computer Science, Lund University, December 1997.
- [7] E. Bjarnason, G. Hedin, K. Nilsson. Interactive Language Development for Embedded Systems. *Nordic Journal of Computing* 6(1999), 36–55.
- [8] A. Cornils and G. Hedin. Statically Checked Documentation with Design Patterns. In *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages (TOOLS Europe 2000)*, St. Malo, France, June 2000. IEEE.
- [9] A. Cornils and G. Hedin. Tool Support for Design Patterns using Reference Attributed Grammars. In D. Parigot and M. Mernik, editors, *Third*

- Workshop on Attribute Grammars and their Applications, WAGA'2000*, Ponte de Lima, Portugal, July 2000.
- [10] 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, Ca., July 1986. ACM SIGPLAN Notices, 21(7).
- [11] S. Gestegård. *Emulation Software for Executable Specifications*. Master's thesis. LU-CS-EX:99-6. Dept. of Computer Science, Lund University, Sweden, April 1999.
- [12] G. Hedin. Incremental Attribute Evaluation with Side-effects. In D. Hammer, editor, *Compiler Compilers and High Speed Compilation (2nd CCHSC Workshop)*, pp 175–189, Berlin, GDR, October 1988, LNCS 371, Springer-Verlag.
- [13] G. Hedin. An object-oriented notation for attribute grammars. *ECOOP'89*. BCS Workshop Series, pp 329–345, Cambridge University Press. 1989.
- [14] G. Hedin. Incremental static-semantic analysis for object-oriented languages using Door attribute grammars. In H. Alblas and B. Melichar, editors, *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems*, pp 374–379, Prague, June 1991, LNCS 545, Springer-Verlag.
- [15] G. Hedin. *Incremental Semantic Analysis*. PhD thesis, Department of Computer Science, Lund University, Sweden, March 1992.
- [16] G. Hedin. An overview of door attribute grammars. *International Conference on Compiler Construction (CC'94)*. LNCS 786, Springer Verlag. 1994.
- [17] R. Hoover. *Incremental Graph Evaluation*. PhD thesis, Cornell University, Ithaca, N.Y., May 1987. Tech. Rep. 87-836.
- [18] F. Jalili. A general linear time evaluator for attribute grammars. *ACM SIGPLAN Notices*, Vol 18(9):35–44, September 1983.
- [19] 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.
- [20] 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*, pp 141–151, New Orleans, La., January 1985. ACM.
- [21] L. G. Jones. Efficient evaluation of circular attribute grammars. *ACM TOPLAS*, 12(3):429–462, 1990.
- [22] M. Jourdan. An optimal-time recursive evaluator for attribute grammars. In M. Paul and B. Robinet, editors, *International Symposium on Programming, 6th Colloquium*, LNCS 167, pages 167–178. Springer-Verlag, 1984.
- [23] G. Kaiser. *Semantics for Structure Editing Environments*. PhD thesis, Carnegie-Mellon University, Pittsburgh, Pa., May 1985. CMU-CS-85-131.
- [24] U. Kastens. Ordered attributed grammars. *Acta Informatica*, 13:229–256, 1980.
- [25] U. Kastens and W. M. Waite. Modularity and Reusability in Attribute Grammars. *Acta Informatica*, 31:601–627, 1994.
- [26] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2), 127–145, June 1968.
- [27] 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.
- [28] 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.
- [29] 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.
- [30] E. Magnusson and G. Hedin. Program visualization using reference attributed grammars. In *Proceedings of NWPER'2000, the 9th Nordic Workshop on Programming and Software Development Environment Research*, Bergen, Norway, 2000.
- [31] P. Persson and G. Hedin. Interactive Execution Time Predictions Using Reference Attributed Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA '99*, pp 173–184, Amsterdam, The Netherlands, March 1999.
- [32] P. Persson and G. Hedin. An Interactive Environment for Real-Time Software Development. In *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages (TOOLS Europe 2000)*, St. Malo, France, June 2000. IEEE.

- [33] A. Poetzsch-Heffter. Programming language specification and prototyping using the MAX System. In M. Bruynooghe, J. Penjam, editors, *Programming Language Implementation and Logic Programming*, pp 137–150. LNCS 714, Springer-Verlag, 1993.
- [34] A. Poetzsch-Heffter. Prototyping realistic programming languages based on formal specifications. *Acta Informatica* 34, 737–772 (1997).
- [35] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator. A system for constructing language-based editors*. Springer Verlag, 1989.
- [36] A. Sasaki and M. Sassa. Circular attribute grammars with remote attribute references. In D. Parigot and M. Mernik, editors, *Third Workshop on Attribute Grammars and their Applications, WAGA '2000*, Ponte de Lima, Portugal, July 2000.
- [37] Sun Microsystems. *Inner Classes Specification*. 1996. URL: <http://java.sun.com/products/~jdk/1.1/docs/guide/innerclasses>
- [38] S. A. Vorthmann. *Modelling and Specifying Name Visibility and Binding Semantics*. CMU-CS-93-158. Carnegie Mellon University, Pittsburgh, Pa., July 1993.
- [39] B. Woolf. Null Object. In R. Martin et al. (eds), *Pattern Languages of Program Design 3*, pp 5–18, Addison Wesley, 1998.