Circular Reference Attributed Grammars - their Evaluation and Applications

Eva Magnusson [1]

Dept of Computer Science
Lund University
Lund, Sweden

Görel Hedin [2]

Dept of Computer Science
Lund University
Lund, Sweden

Abstract

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

1 Introduction

Attribute grammars (AGs), as introduced by Knuth [19], allow computations on a syntax tree to be defined declaratively using attributes where each at-

[1] Email: `eva@cs.lth.se`
[2] Email: `gorel@cs.lth.se`

tribute is defined by a semantic function of other attributes in the tree. An attribute is either used to propagate information upwards in the tree (synthesized attribute) or downwards in the tree (inherited attribute). In the original form of AGs, the definition of an attribute may depend directly only on attributes of neighbor nodes in the tree. Furthermore, the dependencies between attributes may not be cyclic. The first of these restrictions is lifted by Reference Attributed Grammars (RAGs) [11] and similar formalisms, e.g., [24], [2]. In these formalisms, an attribute may be a reference to an arbitrarily distant node in the tree, and an attribute may be defined in a semantic function by directly accessing attributes of the reference (remote access). It has been shown earlier how RAGs support the easy specification and automatic implementation of many practical problems, for example, name- and type analysis of object-oriented languages [11], execution time prediction [23], program visualization [22], and design pattern checking [7].

The second of the restrictions mentioned above, circular definitions, is lifted by Circular Attribute Grammars (CAGs) such as those of Farrow [9] and Jones [15]. The traditional AG requirement of non circularity is a sufficient but not necessary condition to guarantee that an AG is well defined in the sense that all semantic rules can be satisfied. It suffices that all attributes involved in cyclic dependencies have a fixed point that can be computed with a finite number of iterations. In CAGs, circular dependencies between attributes are allowed provided that such a fixed point is available for all possible trees. This is guaranteed if the values for each attribute on a cycle can be organized in a lattice of finite height and if all the semantic functions involved in computing these attributes are monotonic on the respective lattices. Several authors (e.g., [9], [15], [26]) have shown how the possibility of circular definitions of attributes allows simple AG specifications for some well-known problems from different areas. Examples include data-flow analysis, code optimizations, and properties of circuits in a hierarchical VLSI design system. Farrow [9] also demonstrates how alternative non-circular specifications in some cases can be constructed with additional huge complexity, including, e.g., the use of higher-order functions. The circular specifications, in contrast, are both easy to read and understand and easy for the AG author to write.

In this paper, we combine Circular Attribute Grammars (CAGs) and Reference Attributed Grammars (RAGs) into Circular Reference Attributed Grammars (CRAGs). We demonstrate how CRAGs can take advantage of both the combined formalisms, making it possible to express many new problems in a concise and straight-forward way. To exemplify, we show how to specify the *nullable*, *first*, and *follow* sets used in parser construction. These sets are traditionally defined using recursive equations and computed imperatively by iteration. We demonstrate in this paper how the recursive definitions can be expressed directly using CRAGs. We also revisit some of the classical examples of CAGs, in particular, constant evaluation and live analysis, and show how their solutions are facilitated by CRAGs. We have developed a general recursive evaluation algorithm for CRAGs and implemented it in our tool JastAdd [12], which is an aspect-oriented compiler construction tool supporting RAGs. For evaluation, we present some experimental results of the CRAG evaluation of the *nullable*, *first*, and *follow* problems as compared to the corresponding handcoded iterative implementation.

There is some previous work on combining RAG-like formalisms with CAGs. Boyland has implemented a similar combination in his APS system [2]. Sasaki & Sassa present Circular Remote Attribute Grammars (also abbreviated CRAGs), which on the surface is similar to our CRAGs [26]. However, Sasaki & Sassa assume that the remote links are computed separately outside the attribute grammar.

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

2 Existing Evaluation Algorithms

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

Traditional AGs are required to be noncircular, but, as has been shown by, e.g., Farrow [9] and Jones [15], grammars with circular dependencies under certain constraints can be considered well defined in the sense that it is possible to satisfy all semantic rules for all possible syntax trees. One way to formulate the constraints is to require that the domain of all attributes involved in cyclic chains can be arranged in a lattice of finite height and that all semantic functions for these attributes are monotonic.[3] The evaluation of circularly defined attributes can be regarded as a special case of solving the equation $X = f(X)$ for the value of X. By giving X the bottom of the lattice as start value the iterative process $X_{i+1} = f(X_i)$ will converge to a least fixed point

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

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

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

for which all involved semantic rules are satisfied.

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

2.1 Detection of circularity

The problems of deciding whether a traditional AG is circular and of identifying the attributes taking part in cycles have been addressed by several researchers. In [19] Knuth developed a polynomial algorithm for circularity testing. The algorithm is conservative, i.e., circularity is always detected but some noncircular AGs may be reported to be circular. Later [20] Knuth constructed an exact algorithm which is exponential in time and space complexity. Rodeh & Sagiv [25] extended these algorithms to deal with the problem of finding circular attributes. They developed a polynomial approximation algorithm, i.e., all circular attributes are discovered but some noncircular attributes may be reported to be circular. They also constructed an exact algorithm with exponential time complexity and showed that finding the circular attributes is a harder problem than testing for circularity. The problem of testing reference attributed grammars for circularity can be addressed as in [26] by introducing all possible remote edges with lots of potential cycles in the dependency graph as a result. Most abstract syntax trees will, however, not contain any cycles and the iterations performed by the generated evaluator are unnecessary. In [2] Boyland has defined an extension to traditional AGs called remote attribute grammars, which supports reference attributes like in RAGs and also additional features like collection attributes that can be written from remote locations. In [5] he shows that testing remote attribute grammars for circularity is undecidable and examines techniques for approximation.

2.2 Evaluation of circular attribute grammars

Jones [15] proposes a dynamic evaluation algorithm derived from the underlying attribute dependency graph. Optimal dynamic evaluation for circular AGs is obtained by analyzing the dependency graph dynamically to identify its strongly connected components. A strongly connected component is a maximal set of vertices in which there is a path from any one vertex in the set to any other vertex in the set. All attribute instances belonging to the same strongly connected component are thus dependent of each other. Each strongly connected component is contracted into a single node to obtain a new graph `C(G)`, which is acyclic and can be ordered topologically and evaluation can follow this order. A vertex in `C(G)` corresponding to more than one vertex in the original graph represents a set of attribute instances that are all dependent of each other and they will be evaluated in a single fixed-point evaluation. The graphs must be constructed initially. When the attribute grammar is acyclic, Jones' algorithm reduces to a standard optimal algorithm for noncircular evaluation. His scheme is not immediately applicable to RAGs since the reference attributes introduce dependencies that are not known until they have been evaluated.

Farrow [9] introduced a static evaluation technique based on the one by Katayama [18], but modified to compute the fixed point for attributes which potentially have circular dependencies. His scheme is also limited to traditional AGs without remote references since it depends on deriving the attribute dependencies statically from the productions of the grammar. Sasaki & Sassa [26] have elaborated on the technique of Farrow in the presence of remote references. However, these references are not considered to be a part of the attribute grammar and must be evaluated separately in an initial phase. They also make the additional assumption that cycles do not appear without remote references, a constraint that facilitates the check for convergence.

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

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

The static techniques of Farrow as well as that of Sasaki & Sassa have another shortcoming in that iterative evaluation will include a possibly large number of noncircular attribute instances below the AST node associated with the circularly defined attributes that started the iterative process. Case (III) in Fig. 2 is an example. The noncircular attributes b, c and d will then be evaluated during each iteration of the evaluation of component A.

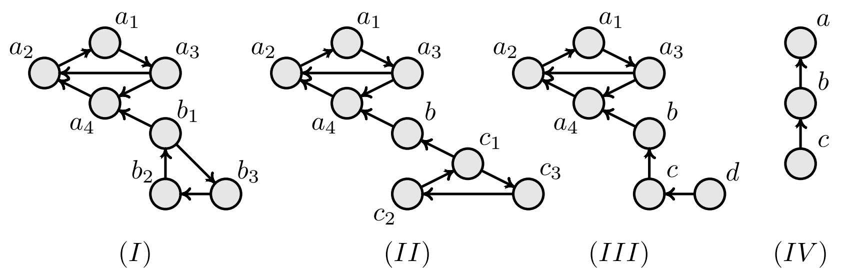


Fig. 2. Different cases of dependencies involving strongly connected components.

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

2.3 Demand-driven evaluation of AGs

We will base the evaluation of CRAGs on a general demand-driven evaluator for non-circular AGs where each attribute is implemented by a method that recursively calls the methods implementing other attributes. By caching evaluated attribute values in the syntax tree, the evaluator is optimal in that it evaluates each attribute at most once. (Our experimental system allows the user to choose which attributes are to be cached. In the rest of this paper we will, however, assume that all attributes are cached in order to achieve optimality.) Circular dependencies can be checked at evaluation time by keeping track of which attributes are being evaluated. In principle, this evaluator is the same as the ones used for traditional AGs by Madsen [21], Jalili [13], and Jourdan [16], although we use an object-oriented implementation [12]. The evaluator is dynamic in that dependencies are not analyzed statically. In fact, the dependencies between attributes need not be analyzed at evaluation time either since the call structure of the recursive evaluation automatically results in an evaluation in topological order. Thus, in contrast to other dynamic evaluation algorithms, no explicit dependency graph is built. The demand-driven dynamic algorithm is sufficiently fast for practical use, also for large examples. Current experiments with generated Java compilers indicate that this kind of evaluation method easily performs within a factor of 3 from hand-written compilers [8]. Our evaluator is implemented in Java which provides a straight-forward implementation of the algorithm. Fig. 3 shows a fragment of an AG based on abstract syntax and the corresponding evaluator code in Java.

AG	Evaluator code
	abstract class Node { Node ancestor; }
Exp { **syn** int val; }	**abstract class** Exp **extends** Node { int val_value; boolean val_computed = false; boolean val_visited = false; **abstract** int val(); }
AddExp: Exp ::= Exp$_1$ Exp$_2$ { val = Exp$_1$.val + Exp$_2$.val; }	**class** AddExp **extends** Exp { Exp exp1, exp2; int val() { **if** (val_computed) **return** val_value; **if** (!val_visited) { val_visited = true; val_value = exp1.val() + exp2.val(); val_computed = true; val_visited = false; **return** val_value; } **else throw new** RuntimeException ("Circular definition of attribute"); } }

Fig. 3. Demand-driven evaluator for noncircular AG.

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

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

Inherited attributes are implemented in a similar, although slightly more involved, manner, making use of the ancestor field to call methods of the ancestor node. See [12] for details.

2.4 Demand-driven evaluation of RAGs

RAGs can be evaluated using the same demand-driven algorithm as for AGs with the extension of allowing attributes to be references to other nodes in the AST [12]. A typical use of reference attributes is in name analysis, where applied occurrences of identifiers are linked to declared occurrences. Fig. 4 shows fragments of a typical RAG with such links. For example, the `IdExp` production contains a reference attribute `decl` which is a reference to the appropriate `Decl` node in the AST. The implementation of the evaluator is a straight-forward extension of the demand-driven AG evaluator. An access to a reference attribute is translated to a call to the corresponding method computing the reference value. For example, the `decl()` method in `IdExp` computes the `decl` reference value. This is done by first computing the value of the `env` attribute (also a reference attribute) and then calling the `lookup` method of the `env` object.

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

3 An evaluator for CRAGs

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

RAG	Evaluator code
Any { **inh** Block env; }	**class** Any **extends** Node { Block env() { ... } }
Block: Any ::= Decl* Stmt* { Decl lookup(String name) { ... } }	**class** Block **extends** Any { List decls; List stmts; Decl lookup(String name) { ... } }
Decl: Any::= Type <ID> { }	**class** Decl **extends** Any { Type type; String ID; }
Exp: Any { **syn** Type tp; }	**class** Exp **extends** Any { **abstract** Type tp(); }
IdExp: Exp ::= <ID> { **syn** Decl decl = env.lookup(<ID>); **syn** Type tp = decl != null ? decl.type : null; }	**class** IdExp **extends** Exp { String ID; ... Decl decl() { ... decl_value = env().lookup(ID); ... } Type tp() { ... tp_value = decl() != null ? decl().type : null; ... } }

Fig. 4. Example of RAG and corresponding evaluator.

3.1 Basic algorithm

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

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

CRAG	Evaluator code

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

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

Fig. 5. Example of CRAG fragment and corresponding classes.

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

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

calls for evaluation of attributes on which `s` is dependent, some of which will be in the same cycle as `s`.

3.2 Comparison of algorithms

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

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

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

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

Nested iterative evaluations are avoided

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

Iterative evaluation of definitely noncircular attributes is avoided

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

3.3 Improving the algorithm

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

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

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

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

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

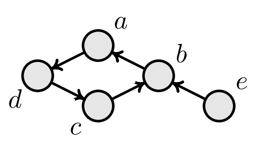
Fig. 7. Pseudo-code for improved evaluation of a definitely noncircular attribute.

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

Sasaki & Sassa [26] also have a refined mostly static version of their originally completely static technique. The basic idea is here to have several versions of attribute evaluation sequences, one for each possible pattern of remote dependency edges. The actual pattern for each subtree in the AST is then computed at runtime and the evaluator selects the proper version. If there are no actually circular attributes in a subtree, iterations are avoided for the production at its root, provided cycles are always caused by remote references. It is not clear if their algorithm can be generalized to deal with cyclic behavior that is not caused by remote links. The refinement deals only with potentially circular attributes that are not actually circular. Their evaluator will still make unnecessary iterations for definitely noncircular attributes in a subtree below AST nodes corresponding to productions with actually circular attributes.

3.4 Robust improved algorithm

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



$$\begin{aligned}a &= b\\ b &= c.union(e)\\ c &= d\\ d &= a\\ e &= \{id\}\end{aligned}$$

Fig. 8. Equations for some attributes creating cyclic dependency.

In order to make the algorithm robust to such grammar errors, the algorithm can be modified as follows. Using the information about which attributes are declared circular, it is possible to keep track of which nodes in the dependency graph might belong to the same strongly connected component. If a visited node belonging to another component is encountered, then an error has ocurred. In the case described above the evaluator would consider attributes a and b to belong to one component and attribute d to another. When the evaluation of d demands a, a visited node belonging to a different component is encountered. The scheme of keeping track of components can be realized by adding vertices of the dependency graph to a set during evaluation, as long as only potentially circular attributes are encountered. This set will be stacked together with the `CHANGE` flag when an iteration is temporarily suspended as in Fig. 7. When a visited node is encountered it can then be checked if it belongs to the set of the component actually being brought to a fixed point. Thus, in case of a missing circular declaration, the algorithm will detect the error, identify the attributes involved, and raise an exception.

3.5 Comparison to related work

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

Boyland takes a similar approach in his APS system where he has implemented support for evaluation of circular attributes for remote attribute grammars. His evaluation method is based on demand-driven evaluation and performs topological evaluation of strongly-connected components, allowing non-monotonic dependencies between components. However, he provides no explicit evaluation algorithms [2].

For ordinary attribute grammars, the dependency graph can be computed from the grammar and constructed before evaluation. The static attribute evaluation algorithms available for ordinary attribute grammars, like OAGs [17], rely on this property in order to compute approximations of the dependency graph before evaluation. The same holds for the static evaluation algorithms for circular attribute grammars, like Farrow's algorithm [9].

The development of static evaluators for subcategories of RAGs and CRAGs is a problem that we have not pursued, but there is some other work in this direction. In [3], [4], [5], Boyland addresses the problem of analyzing (noncircular) non-local dependencies statically. He develops a technique to schedule evaluation statically and shows how it may be implemented incrementally. The technique has been applied to variants of name and type analysis including the static-semantics of a simple object-oriented language.

Sasaki & Sassa [26] allow circular dependencies as well as remote links between nodes in the AST, but links between nodes are not considered a part of the AG and must be provided by a separate initial phase that they have not elaborated further on in their paper. In contrast, our demand-driven evaluation technique allows reference attributes as well as ordinary attributes to be evaluated in the same manner. An additional constraint in the scheme of Sasaki & Sassa is that cycles are assumed to arise only from remote references.

As was discussed in Section 3.2, the static evaluation algorithms of Farrow and of Sasaki & Sassa have suboptimal behavior for strongly connected components of circular attributes, while the dynamic algorithm of Jones [15] is optimal. Jones algorithm computes the strongly connected components for a given AST before evaluation, based on the actual attribute dependencies. For CRAGs, such pre-evaluation computations are not possible in general, since the reference attributes give rise to attribute dependencies that are not known until after the evaluation of those reference attributes. For this reason, the strongly connected components in CRAGs cannot, in general, be computed before evaluation. Our evaluation algorithm detects strongly connected components during evaluation, and does not always have sufficient information to distinguish between two components. In this case, the components will be evaluated together rather than in topological order, thereby yielding suboptimal evaluation. However, in Section 3.3 we showed how to avoid many of these cases by using cached attribute values. The remaining suboptimal case is the one where there are two adjacent strongly connected components. I.e., where an attribute in one component depends directly on the attribute in another component, like in case (I) of Fig. 2. To handle also these cases, there would be the need for either more information given by the grammar author, or for some kind of approximative pre-evaluation analysis. The development of such analyses remains an open issue. For non-circular RAGs, dynamic demand-driven evaluation using caching is optimal (each attribute is evaluated at most once).

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

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

Our evaluator presently does not check whether circularly defined attributes take their values from a lattice of finite height or if their defining semantic functions are monotonic. Thus there is no guarantee that iterations will converge. Our approach is in this respect similar to that of, e.g., Farrow [9] and means that we rely on the AG author to ensure that the semantic functions involved are properly constrained.

4 Application Examples

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

4.1 Computation of nullable, first and follow

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

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

Fig. 9 shows an example context-free grammar and its values for *nullable*, *first*, and *follow* (grammar 3.12 in Appel [1]).

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

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

Computation of nullable

The following equations hold for *nullable*:

(i) Let X be a nonterminal with the productions $X \rightarrow \gamma_1, X \rightarrow \gamma_2, \ldots X \rightarrow \gamma_n$

X is nullable if any of its production right-hand sides is nullable:

$nullable(X) == nullable(\gamma_1) \,||\, nullable(\gamma_2), ... \,||\, nullable(\gamma_n)$

(ii) Let ϵ be an empty sequence of terminal and nonterminal symbols.
The empty sequence is nullable:

$nullable(\epsilon) == true$

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

$nullable(\gamma) == nullable(s) \,\&\&\, nullable(\delta)$

(iv) A terminal symbol t is not nullable:

$nullable(t) == false$

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

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

Computation of first

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

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

$first(X) == first(\gamma_1) \cup first(\gamma_2) ... \cup first(\gamma_n)$

(ii) Let ϵ be an empty sequence of terminal and nonterminal symbols.

$first(\epsilon) == \emptyset$

(iii) Let $s\delta$ be a nonempty sequence of terminal and nonterminal symbols where s is the first symbol and δ is the remaining (possibly empty) sequence.

$first(s\delta) ==$ **if** $(nullable(s))$ **then** $first(s) \cup first(\delta)$ **else** $first(s)$

(iv) Let t be a terminal symbol.

$first(t) == \{t\}$

The equation system is circular which is evident from (i) since X might be identical to, or derivable from, one of the nonterminal symbols on the right-hand side of one of the productions. We can also note that the equations

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

Fig. 10. A CRAG that computes *nullable*.

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

Computation of follow

The definition and CRAG for *follow* is similar in style to *nullable* and *first*, but makes additional use of reference attributes: To compute *follow* for a nonterminal X we need to locate all the applied occurrences of X and look at the subsequent symbols. To this end, reference attributes are used for linking an `NDecl` to all its `NUses`. The additions of such cross-referencing attributes are straight-forward using RAGs, for example by defining a set of `NUse` references at each `NDecl`. These set-valued attributes can easily be defined using parameterized attributes which are analogous to virtual functions, and which are available in RAGs [11]. In essence, such a function simply tra-

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

Fig. 11. A CRAG that computes *first*.

verses a suitable portion of the AST to collect the appropriate information. An example of such a computation is shown in our paper on using RAGs for visualization computations [22]. The collection attributes of Boyland [2] would provide a more elegant way of defining such cross-reference information. With these cross-reference attributes in place, the specification of $follow$ becomes as straight-forward as for *nullable* and *first*, and is also circular.

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

Experimental results

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

The results are shown in Fig. 12. The grammars Appel 1 and Appel 2 are small example grammars from [1]. Appel 1 is a toy language (the same as in Fig. 9.) with 3 nonterminals (#N) and 6 productions (#P) and Appel 2 is a grammar for simple arithmetic expressions. Tiny is a grammar for a small block-structured language. The grammar for Java 1.2 is the largest and has been taken from the examples distributed with JavaCC [14]. It has about 160 nonterminals when written in our CFG language. The times given are average times for 100 executions on a Sun Ultra 80 using the HotSpot JVM. The results indicate that the evaluator of CRAG performs as well as the handwritten iterative evaluation code. For a large grammar like Java the declarative approach even seems to be superior. One explanation could be that in an imperative style fixed-point iteration, the order in which the productions are processed is very important. (See, e.g. [1] chapter 17.4.) The CRAG evaluator, on the other hand, traverses the dependency graph depth first, i.e., in topological order, and the iterations will thus usually converge faster.

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

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

Fig. 12. Computation of *nullable*, $first$, and $follow$ for some different grammars.

4.2 Using constants before declaration

This is an example described by Farrow in [9] and deals with a language where constants can be defined in terms of other constants and where use of a constant before its declaration is legal as in the following example:

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

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

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

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

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

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

The `val` attribute will be noncircular exactly when Farrow's cyclic specification produces a complete table of constant values, i.e., when each constant has a defining expression and no constant is defined in a circular manner.

```
ConstDecl ::= IdDecl Exp { syn Integer val = Exp.val; }
Exp { syn Integer val; }
AddExp : Exp ::= Exp_1 Exp_2 { val = Exp_1.val + Exp_2.val; }
...
ConstUse : Exp ::= <ID> {
    syn ConstDecl decl = ...;
    val = decl != null ? decl.val : undefined;
}
IntExp : Exp ::= <INT> { val = <INT>; }
```

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

Should some constants be part of circular definitions like in the example above, this is a programming error that should be caught by the compiler. To use circular attributes is not an option here since there does not, in general, exist any fixed-point solution. For the CRAG above, the evaluator will throw an exception when it discovers the circular dependency. This is, of course, not an acceptable behavior for a production compiler. An improved CRAG can instead check explicitly for such erroneous circular definitions by introducing an attribute `wellDefined` in class `ConstDecl`. Its value can be specified in a noncyclic manner by a recursive function that builds the set of all constants of which a certain constant is dependent and checks that the constant is not itself in this set. An alternative way is to introduce a circular boolean attribute `wellDefined` in `ConstDecl` and `Exp` with bottom value `false`. The specification of this attribute is straight-forward. A simple integer expression (`IntExp`) is well-defined and a compound expression is well-defined if all its subexpressions are well-defined.

4.3 Live analysis in optimizing compilers

One of the most frequently used examples of cyclic dependencies in AGs is the performance of live analysis for variables. Farrow [9], Jones [15], and Sasaki & Sassa [26] all focus on this example in their papers.

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

Farrow and Jones both exemplify with a language with loop-structures like `for`- and `while`-statements. No reference attributes need to be involved here, but the specifications become cyclic for loop-statements. Sasaki & Sassa use a smaller language with only assignment statements, `label`-statements and `goto`-statements. Here remote attributes are used to link `goto` nodes in the AST to their corresponding `label` nodes. Cycles can in their simple example language arise only if a program contains `goto`-statements. Their evaluation technique (or rather the technique to check for convergence) requires that cycles always include remote links. This means that the evaluation process described in [26] would not directly be capable of handling, e.g., a language with structured loop-statements without adding explicit remote links. Also, as has been mentioned before, their links between `gotos` and `labels` in the AST are not ordinary attributes, but need to be provided by a separate phase that takes place before attribute evaluation.

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

5 Conclusions

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

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

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

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

Future work includes further improvements of the evaluator. One idea we are looking at is how to isolate strongly connected components by modularizing the grammar. Such modularization is natural to do anyway from a grammar writing perspective, and can probably be used for improving the evaluator performance and for allowing non-monotonic dependencies between components in different modules. We are also looking at techniques for automatically deciding which attributes to cache to provide best performance and memory usage. It would be desirable to let the user decide what attributes to save in the AST nodes and let the tool help to decide when to cache other attributes temporarily to avoid inefficiencies and for check of convergence. One idea could be using a cache like in [26]. We also plan to apply CRAGs to more problem areas. In [8] a formalism for rewriting abstract syntax trees is presented. The formalism, Rewritable Reference Attributed Grammars (ReRAGs), has been implemented in our aspect-oriented compiler tool JastAdd. We plan to merge this extension of JastAdd with our CRAG extension. We believe that there are problem areas where this combination would prove useful.

Acknowledgements

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

References

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

[2] Boyland, J. T., *Descriptional Composition of Compiler Components.* Ph.D. thesis, University of California, Berkeley, California, 1996.

[3] Boyland, J. T., *Analyzing Direct Non-Local Dependencies in Attribute Grammars.* In Proceedings of CC´98: International Conference on Compiler Construction, 31-49. LNCS 1383, Springer-Verlag, 1998.

[4] Boyland, J. T., *Incremental evaluators for remote attribute grammars.* In Electronic Notes in Theoretical Computer Science **65(3)**. June, 2002.

[5] Boyland, J. T., *Remote Attribute Grammars.* Manuscript, 2003.

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

[7] Cornils, A., and G. Hedin, *Tool Support for Design Patterns based on Reference Attributed Grammars.* Proceedings of WAGA´00, Workshop on Attribute Grammars and Applications. Ponte de Lima, Portugal, July 2000.

[8] Ekman, T., and G. Hedin, *Rewritable Reference Attributed Grammars.* In Proceedings of ECOOP 2004: 18th European Conference on Object-Oriented Programming. Oslo, Norway, 2004.

[9] Farrow, R., *Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars.* Proceedings of the SIGPLAN´86 Symposium on Compiler Construction, 85-98. Palo Alto, California ACM SIGPLAN Notices **21(7)** (1986).

[10] Hedin, G., *An object-oriented notation for attribute grammars.* ECOOP´89. BCS Workshop Series, 329-345, Cambridge University Press, 1989.

[11] Hedin, G., *Reference Attributed Grammars.* Informatica **(24)** (2000), 301-317. Slovenia.

[12] Hedin, G., and E. Magnusson, *The JastAdd system - an aspect-oriented compiler construction system.* SCP - Science of Computer Programming, **47(1)** (2002), 37-58. Elsevier.

[13] Jalili, F., *A general linear time evaluator for attribute grammars.* ACM SIGPLAN Notices, **18(9)** (1983), 35-44.

[14] JavaCC. URL: `http://www.webgain.com/products/java_cc/`.

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

[16] Jourdan, M., *An optimal-time recursive evaluator for attribute grammars.* In M. Paul and B. Robinet, editors, International Symposium on Programming, 6th Colloquium, Lecture Notes in Computer Science **167**, (1984), 167-178. Springer-Verlag.

[17] Kastens, U., *Ordered Attributed Grammars.* Acta Informatica, **13** (1980), 229-256.

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

[19] Knuth, D. E., *Semantics of context-free languages.* Mathematical Systems Theory, **2(2)**, (1968), 127-145.

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

[21] Madsen, O. L., *On defining semantics by means of extended attribute grammars.* Semantics-Directed Compiler Generation, LNCS **94**, (1980), 259-299, Springer-Verlag.

[22] Magnusson, E., and G. Hedin, *Program Visualization using Reference Attributed Grammars.* Nordic Journal of Computing **7**, (2000), 67-86.

[23] Persson, P., and G. Hedin, *Interactive Execution Time Predictions Using Reference Attributed Grammars.* In WAGA'99, Second Workshop on Attribute Grammars and their Applications. Amsterdam, The Netherlands, March 1999.

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

[25] Rodeh, M., and M. Sagiv, *Finding Circular Attributes in Attribute Grammars.* JACM - Journal of the ACM, **46(4)** (1999), 556-575.

[26] Sasaki, A., and M. Sassa, *Circular Attribute Grammars with Remote Attribute References.* Waga'00, Third Workshop of Attribute Grammars and their Applications. Ponte de Lima, Portugal, July 2000.