Termination Analysis for Higher-Order Attribute Grammars*

Lijesh Krishnan and Eric Van Wyk

Department of Computer Science and Engineering
University of Minnesota, Minneapolis MN, USA,
krishnan,evw@cs.umn.edu

Abstract. This paper describes a conservative analysis to detect non-termination in higher-order attribute grammar evaluation caused by the creation of an unbounded number of (finite) trees as local tree-valued attributes, which are then themselves decorated with attributes. This type of non-termination is not detected by a circularity analysis for higher-order attribute grammars. The analysis presented here uses term rewrite rules to model the creation of new trees on which attributes will be evaluated. If the rewrite rules terminate then only a finite number of trees will be created. To handle higher-order inherited attributes, the analysis places an ordering on non-terminals to schedule their use and ensure a finite number of passes over the generated trees. When paired with the traditional completeness and circularity analyses and the assumption that each attribute equation defines a terminating computation, this analysis can be used to show that attribute grammar evaluation will terminate normally. This analysis is applicable to a wide range of common attribute grammar idioms and has been used to show that evaluation of our specification of Java 1.4 terminates.

1 Introduction

Silver [14] is an extensible attribute grammar system designed to support the modular specification of languages. Silver specifications define host and extension concrete syntax as well as domain-specific semantics such as optimizations, transformations, and error-checking. It has been used to write extensible grammars for mainstream languages such as Java 1.4 [15]. Attribute grammars (AGs) were introduced by Knuth in 1968 [8] as a means to assign semantics to syntax trees, by associating tree nodes with named values known as attributes. Once the parser constructs the program syntax tree, the attribute evaluator evaluates its undefined instances one at a time. In 1989 Vogt *et al.* introduced higher-order attribute grammars (HOAGs) [17] in which syntax trees can be computed and passed around a syntax tree as attribute values. These trees may also be decorated with attributes which are then evaluated.

Static analyses of attribute grammars detect problems before evaluation and ensure that the attribute grammar is well-defined. According to Vogt *et al.*,

* This work is partially supported by NSF Awards No. 0905581 and No. 1047961.

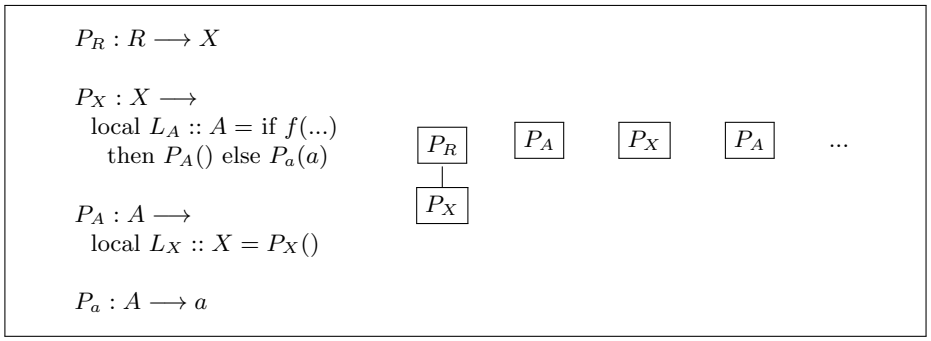

Fig. 1. A higher-order attribute grammar (specified in [17]) for which tree creation may not terminate, and an example of a non-terminating tree creation sequence.

a *well-defined* higher-order attribute grammar is one that: is *complete* so that every synthesized, inherited and local instance on a node has a definition; is *non-circular* so that no attribute value depends on itself, either directly or indirectly; and has a *finite number of new trees* created during attribute evaluation. For the first two conditions, Vogt *et al.* specify extended versions of Knuth's tests for completeness and circularity, which prevents abnormal termination of attribute evaluation. But unlike with attribute grammars without higher-order attributes, attribute evaluation for a complete, non-circular HOAG may not terminate since creating an unbounded number of trees during higher-order attribute evaluation can lead to non-termination. For example, Fig. 1 gives a grammar (borrowed from [17]) for which an infinite number of trees may be created. This explains why Vogt *et al.* included the third condition in their definition of well-definedness. (However, "well-defined" is often used to describe grammars that meet only these first two criteria: completeness and non-circularity. Vogt's Ph.D. dissertation adopts this use of the term [16].)

Circularity analysis prevents, among other problems, specifications that would cause the construction of infinite syntax trees, but it does not prevent the creation of an infinite number of *finite* trees, as seen in the example in Fig. 1. Any analysis to detect the creation of an unbounded number of trees will be conservative; consider the conditional expression in the example. An analysis to guarantee that no evaluation sequences exist with infinitely many tree creation steps, combined with the completeness and circularity tests, would be sufficient to ensure higher-order attribution termination. While Vogt *et al.* describe a condition required to ensure non-termination, it does not seem to have been implemented or evaluated. The analysis presented here is the first, to our knowledge, that uses the structure of the equations and not just the attribute dependencies imposed by them in an analysis for termination of tree creation.

In this paper, we fill this gap with a conservative analysis to ensure that tree creation during attribution terminates. This analysis uses rewrite rules and non-terminal orderings to model tree construction and check for the possibility of

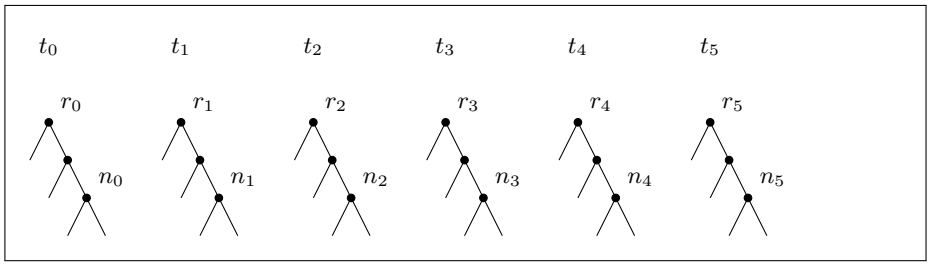

Fig. 2. A tree creation sequence from the original tree t_0, in which each non-initial tree t_{i+1} is created as a local attribute on its predecessor t_i at node n_i.

non-termination of tree construction. The analysis is on a restricted, but useful HOAG class, and assumes a general evaluation model. We use existing termination analyses on term rewriting systems to show termination of higher-order attribute evaluation. The restrictions and the assumption that the grammar is complete and non-circular mean that for any improper evaluation sequence, there is an infinite tree creation sequence starting from the original tree, in which each non-initial tree is created as a local attribute on its predecessor. In Fig. 1 an example of a non-terminating tree creation sequence is shown for that example grammar. Fig. 2 shows this diagrammatically; here the tree t_{i+1} with root node r_{i+1} is created on node n_i of tree t_i rooted at node r_i. Showing all such sequences terminate would ensure that evaluation terminates normally.

We define a procedure to generate, for an attribute grammar, a set of rewrite rules that model local tree creation during attribution, in the absence of inherited attributes. Thus for each grammar, we generate rules that rewrite trees to the values of local higher-order attributed trees that may be created during attribute evaluation. That is, any sub-tree rooted at n_i can be rewritten by the rules to any tree t_{i+1} that might be the value of one of its local higher order attributes. In other words,

$$\langle \text{sub-tree rooted at } n_i \rangle \Longrightarrow^+ t_{i+1}$$

For example, the (non-terminating) rules generated for the grammar in Fig. 1 are as follows:

$$\{\ P_X() \Longrightarrow P_A(),\ P_X() \Longrightarrow P_a(a),\ P_A() \Longrightarrow P_X()\ \}$$

We can show that for any tree creation sequence arising from the evaluation of higher-order attributes, we can construct a rewrite sequence of the same length. Thus if the rules terminate, no infinite tree creation sequence exists. We thus have a guarantee of termination of tree creation, and thereby of attribute evaluation, in the absence of higher-order inherited attributes. We use existing tools such as AProVE [5] to verify that the rules terminate.

The problem of non-terminating higher-order *inherited* attribute evaluation is handled by ordering the grammar non-terminals so that the type of each inherited higher-order attribute value is "smaller" than the non-terminal it decorates.

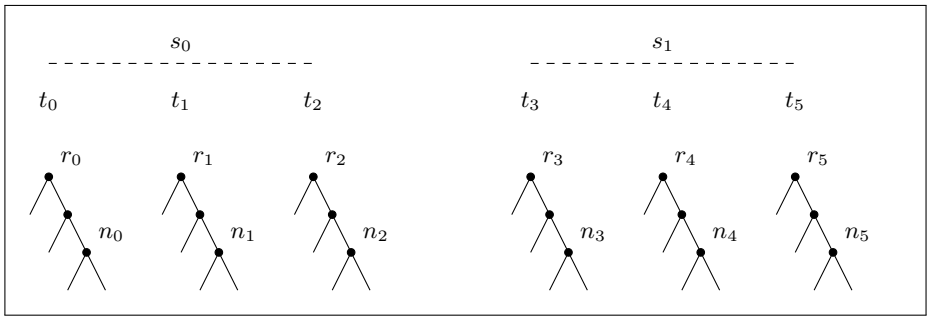

Fig. 3. A tree creation sequence from the original tree t_0, in which each non-initial tree t_{i+1} is created as a local attribute on its predecessor t_i. The sequence consists of a sequence of sub-sequences marked by the use of inherited attributes.

In all other cases of higher-order attribute evaluation, the attribute type is no "larger" than the non-terminal of the node on which it was created. The existence of such an ordering ensures that the number of higher-order inherited accesses in any tree creation sequence is finite. An infinite tree creation sequence such as that shown in Fig. 1 and modeled in Fig. 2 consists of a sequence of tree creation sub-sequences in which the non-terminals at the roots of the trees in each sub-sequence, are of the same "size", as shown in Fig. 3. Within such "constant" tree creation sub-sequences, the rewrite rules can still be used to model tree creation, so that if the rules terminate, then every such constant sub-sequence must terminate. This implies, once again, that no infinite tree creation sequence exists. Thus if the grammar's rules are terminating and its non-terminals can be ordered as desired, then attribute evaluation will terminate.

Contributions and Outline: Our primary contribution is an analysis that detects potential non-termination of tree construction in higher-order attribute grammars. This analysis is based on rewrite rules and an ordering on grammar non-terminal symbols to ensure termination of tree construction and is, to the best of our knowledge, the first of its kind. We also evaluate this analysis on the ABLEJ specification [15], an attribute grammar specification for Java 1.4.

This paper is structured as follows. Section 2 presents background material on higher-order attribute grammars and describes the restricted class of grammars handled by the termination analysis. Section 3 describes how non-termination of attribute evaluation is equivalent to disproving the existence of infinite local tree creation sequences. Section 4 describes how term rewriting rules can be used to model tree creation in the absence of higher-order inherited attributes. Section 5 describes how the analysis handles higher-order inherited attributes by constructing an ordering on the non-terminals that limits the number of inherited accesses. A discussion and evaluation are provided in Section 6. Section 7 describes related work. Finally, Section 8 discusses future work and concludes.

2 A Restricted Class of Higher-Order Attribute Grammars

In this section we describe RHOAGs, the class of higher-order AGs handled by our termination analysis. The design of this class is driven by two main goals. First, it is simple enough to conveniently illustrate the main ideas behind the analysis, but still includes most of Silver's important features and is expressive enough to specify rich, complex grammars. Our attribute grammar for Java 1.4, ABLEJ [15], was converted to this form with a few modest modifications. Second, the restrictions allow us to reduce the termination problem to that of disproving infinite local tree creation sequences. Many features in Silver (such as forwarding [13]) can be translated into this form without a loss of expressivity.

In Fig. 4, we give a formal definition of the attribute grammars in RHOAGs. As in standard AGs [8], there are restrictions on which attribute occurrences may be defined on a given production. Synthesized and local occurrences are defined on their node's production. Inherited occurrences are defined on the production of their node's parent, unless their node is the root of a tree evaluated as a local attribute. In the latter case, the inherited occurrence is defined on the production of the local's parent node on which the local was evaluated.

Unlike standard grammars, definitions and expressions in RHOAGs are further restricted in the following ways. There are no inherited occurrences on the root node, for simplicity. A production may not refer to its own synthesized occurrences, or its children's inherited occurrences; since it computes these values there is no need to do so. Attributes may only be accessed on a node's children, its locals, or its own tree (in the case of inherited attributes). All other kinds of accesses, such as nested accesses of attributes on attributes, are disallowed by the syntax, allowing us to isolate the creation of new syntax trees during evaluation to local attribute evaluation. Functions are primitive and do not take trees as arguments or return them as results.

We use a number of additional terms and notations that we briefly describe here. Trees are typically referred to in this analysis by their root nodes, which are elements of the set N of tree nodes. Attribute instances have the type $N \times A$. We write $n\#a$ for an instance of attribute a on node n in some tree. A *tree term* or simply *term* is a textual representation of a tree that consists of the production name constructing its root node and the terms of its child trees enclosed in parentheses and separated by commas. This is similar to the conventions of algebras and term-algebras. For example, "$P_R(P_X())$" is the term denoting the first tree in Fig. 1. We use $term(n)$ to denote the term representing a tree rooted at node n. The set of terms is denoted by $Term$. An *attribution* of a tree, Γ, maps attribute instances to their values. An attribute value may be an element of the set PV of primitive values, a tree term in $Term$, a tree node in N, or be (as of yet) undefined, in which case it is denoted by $\perp$.

Fig. 5 gives an informal description of the process of higher-order attribute evaluation for a given syntax tree t. Attribute evaluation starts with a syntax tree, usually constructed by a parser, with all of its attribute instances undefined ($\perp$). Attribution begins by evaluating attributes whose definitions are constants,

$G = \langle NT, T, P, S \rangle$ is a context-free grammar.

- NT is a set of non-terminals.
- T is a set of terminals.
- P is a set of productions with signatures of the form $NT \to (NT \cup T)^*$.
- $S \in NT$ is a start non-terminal.

$AG = \langle G, PT, PV, A, @, F, D \rangle$ is an attribute grammar.

- PT is a set of primitive types.
- PV is a set of primitive values with types in PT.
- $A = A_S \cup A_I \cup A_L$ is a set of attributes.
  - A_S is a set of synthesized attributes of type $NT \cup T \cup PT$.
  - A_I is a set of inherited attributes of type $NT \cup T \cup PT$.
  - A_L is a set of local attributes of type NT.
- $@ \subseteq ((A_S \cup A_I) \times NT) \cup (A_L \times P)$ is the occurs-on relation.
- F is a set of primitive functions with type signatures of the form $PT^* \to PT$.
- D returns the set of attribute definitions associated with a given production.

For a production $p \in P$ with signature $X_0 \longrightarrow X_1 ... X_{n_p}$, $D(p)$ contains definitions of the form

- $X_0.a_S = E$ where $a_S \in A_S, a_S @ X_0$
- $X_i.a_I = E$ where $1 \leq i \leq n_p, a_I \in A_I, a_I @ X_i$
- $l = E$ where $l @ p$
- $l.a_I = E$ where $l @ p, a_I \in A_I, a_I @ L_i$ and L_i is the type of l

The expressions E on the right-hand sides of these definitions have the form

- X_i where $0 \leq i \leq n_p$
- $X_0.a_I$ where $a_I \in A_I, a_I @ X_0$
- $X_i.a_S$ where $1 \leq i \leq n_p, a_S \in A_S, a_S @ X_i$
- $l.a_S$ where $l @ p, a_S \in A_S, a_S @ L_i$ and L_i is the type of l
- c where $c \in T$
- $q(E_1, ..., E_{n_q})$ where $q \in P$
- $f(E_1, ..., E_{n_f})$ where $f \in F$
- `if` E_C `then` E_T `else` E_E

Fig. 4. Formal definition of the restricted class RHOAG of higher-order AGs.

SET T to { t }.
WHILE there is an evaluable attribute occurrence $n\#a$ in a tree in T

- **IF** a is a synthesized or local attribute
  - **SET** e to $n\#a$'s defining expression e, specified in n's production.
- **ELSE**
  - **SET** e to $n\#a$'s defining expression e, specified in n's parent node's production.
- **SET** v to the evaluated value of e.
- **IF** a is a local attribute
  - **ADD** an attributed version of the tree term v to T.
  - **SET** $n\#a$ to the root of this tree.
- **ELSE**
  - **SET** $n\#a$ to v.

Fig. 5. An informal description of the process of attribute evaluation process for higher-order attribute grammars.

or which depend solely on attributes on terminal symbols (such as *lexeme*) which are set by the parser. In each evaluation step, an undefined *evaluable* occurrence (*i.e.*, one whose required attribute instances have all been evaluated) is selected and evaluated. Instances are evaluated one at a time until there are no more undefined evaluable attribute instances. If the grammar passes the circularity and completeness tests, the process will end only when all attribute instances have been evaluated. We assume that during attribute evaluation, attribute accesses, function calls and tree-creating steps terminate atomically with valid values. Conditional expressions are evaluated lazily as expected.

The semantics of local attribute evaluation is different from the semantics of other kinds of higher-order attribute evaluation (i.e., synthesized or inherited occurrences). When a local instance is evaluated, the computed tree term value is converted into a full-fledged syntax tree with its own (undefined) attribute instances. This new tree is added to the set of trees that defines each evaluation state. Further, new trees are added only as local attributes due to syntactic restrictions. Attribute evaluation terminates only when all instances on all trees are defined. When an unbounded number of trees are created there are an unbounded number of attribute instances. This leads to the type of non-termination that is detected by the analysis presented here and not detected by higher-order circularity analyses.

A *proper evaluation sequence* terminates with a valid attribution to every attribute instance in the original tree, and any tree created during evaluation. An *improper evaluation sequence* either terminates abnormally due to absent definitions, circularities in attribute definitions, non-terminating function calls,

or other errors in the grammar definition; or does not terminate due to the creation of an infinite number of trees. Our evaluation model is more general than most standard evaluators, including Silver's. This means that for a given syntax tree, while any attribute evaluation sequence possible in Silver is also included in this model, there are sequences in this model that are not possible in Silver. Silver, and other current systems like JastAdd [4] and Kiama [12], use a *demand-driven* attribute evaluation algorithm in which only the attribute values that are needed are computed. The model presented here is simpler but more general and includes all possible models of attribute grammar evaluation. It is therefore possible that for an input tree, evaluation is non-terminating in this model, while it is terminating in Silver.

3 Reducing Infinite Evaluation to Infinite Tree Creation

We can reduce the problem of guaranteeing termination of attribution to that of disproving the existence of infinite tree creation sequences.

> Theorem 1: For a tree in a complete, non-circular grammar with terminating functions, if there is an improper evaluation sequence, then there is an infinite tree creation sequence.

For a complete, non-circular grammar with terminating functions, any evaluation sequence that does not terminate normally contains an infinite number of local tree creation steps. This is because at every step of a valid evaluation sequence, either every tree is attributed, or there is an undefined evaluable attribute occurrence (because the definitions are non-circular). This means that an evaluation sequence that does not terminate normally has an infinite number of evaluation steps. It must further evaluate an infinite number of attribute instances, since in every step, an undefined instance is evaluated. As each tree has only a finite number of attribute instances, an infinite number of trees must be created via local creation steps.

The syntax trees at each evaluation step can be organized into a tree of trees, known as the *tree of locals* (TOL). Each node of a TOL represents a syntax tree. The root node represents the original program syntax tree. A TOL node's child nodes are the local trees that have been created on any of the nodes of its own syntax tree. A path in a TOL is a *tree creation sequence*, i.e., a sequence of trees starting from the program syntax tree, in which each non-initial tree is the value of a local that has been evaluated on its predecessor. An evaluation state's syntax trees can be organized into a TOL since every local is created on an existing tree and every local is evaluated only once. The TOL of the initial evaluation state contains one node, labeled with the program syntax tree. The TOL of each later step either is the same as that of its previous step, or has one extra node and edge. In every local-evaluating step, a node for the new syntax tree is added to the TOL, as a child of the TOL node corresponding to the local's parent syntax

tree. In all other steps, the TOL stays the same, though the attribution to an instance of an existing syntax tree changes. The sequences of trees in Fig. 1 and Fig. 2 are tree creation sequences and are paths in the TOL.

Thus for an evaluation sequence with an infinite number of local tree creation steps, we can construct an infinite trees of locals, with increasing numbers of nodes and edges. Each TOL node has a finite number of children, as each syntax tree has a finite number of AST nodes, each with a finite number of locals. König's Lemma states that an infinite tree in which each node has a finite number of children, has an infinite path. Thus corresponding to an infinite sequence in a TOLs with increasing numbers of nodes and edges, we can construct an infinite path of trees starting from the original tree, in which each non-initial tree is created as a local on its predecessor. Theorem 1 is proved formally in [9] and is the topic of much of the rest of this paper.

4 Modeling Tree Creation with Rewrite Rules

We use rewrite rules to model tree creation so that termination of the rewrite rules implies termination of all tree creation sequences. For a given grammar, we derive a set of rewrite rules based on the higher-order attribute definitions in its productions. Each generated rule is of the form $p(x_1, ..., x_{n_p}) \Longrightarrow r$ where

- p is a production in P with signature $X_0 \longrightarrow X_1 ... X_{n_p}$,
- $x_1, ..., x_{n_p}$ are rewrite rule variables, and
- the rule's right-hand side r contains production names, terminal symbols and rewrite rule variables.

For a production $p \in P$ with signature $X_0 \longrightarrow X_1 ... X_{n_p}$, we generate a set of rules of the form $\{\ p(x_1, ..., x_{n_p}) \Longrightarrow r \mid r \in R(e)\ \}$ for every expression e that is the RHS of a higher-order definition in $D(p)$. Here $R(e)$ returns a set of rule right-hand sides for a given higher-order expression e, as defined below:

e	$R(e)$
X_i	$\{x_i\}$
$X_i.a_S$	$\{x_i\}$
$X_0.a_I$	$\{\texttt{INH}\}$
$l.a_S$	$R(e_L)$ where $(l = e_L) \in D(p)$
$q(e_1, ..., e_{n_q})$	$\{q(r_1, ..., r_{n_q}) \mid \ r_i \in R(e_i),\ 1 \leq i \leq n_q\}$
c	$\{c\}$
`if` e_C `then` e_T `else` e_E	$R(e_T) \cup R(e_E)$

A set of rewrite rules is created for each production in the grammar, based on its attribute definitions. Each rule has a different RHS based on the RHS of the higher-order attribute definitions. Explicit tree creation sub-expressions containing production names, terminal symbols and signature variables are represented as such in the rules' right hand sides. Conditional expressions are handled by generating separate rules for each sub-expression, which means multiple rules may

```
synthesized attribute pp :: String occurs on Stmt, Expr ;
production doWhile s::Stmt ::= s1::Stmt e::Expr {
   s.pp = "do " ++ s1.pp ++ " while ( " ++ e.pp ++ " );";
   local attribute fs::Stmt = consStmt (s1, while (e, s1));
}
production ifThen s::Stmt ::= e::Expr s1::Stmt {
   s.pp = "if ( " ++ e.pp ++ " ) " ++ s1.pp;
   local attribute fs::Stmt = ifThenElse (e, s1, emptyStmt ());
}
production while s::Stmt ::= 'while' '(' e::Expr ')' s1::Stmt { ... }
production consStmt s::Stmt ::= s1::Stmt s2::Stmt { ... }
production assign  s::Stmt ::= id::Id_t '=' e::Expr ';' { ... }
production ifThenElse
  s::Stmt ::= 'if' '(' e::Expr ')' s1::Stmt 'else' s2::Stmt { ... }
production emptyStmt s::Stmt ::= ';' { ... }
```

- `doWhile` $(s_1,\ e) \Longrightarrow$ `consStmt` $(s_1,$ `while` $(e,\ s_1))$
- `ifThen` $(e,\ s_1) \Longrightarrow$ `ifThenElse` $(e,\ s_1,$ `emptyStmt` $())$

Fig. 6. A fragment of a higher-order attribute grammar, and the rules generated to model tree creation during evaluation.

therefore be generated for a single definition or sub-expression. Accesses to synthesized occurrences on a child are represented by the child's signature variable. Accesses to synthesized occurrences on local attributes are replaced by rewrite sub-terms generated from the local's definition. Local attributes are thereby inlined when generating rewrite rules; this is possible for non-circular attribute grammars. Function symbols are not present in higher-order sub-expressions. The rules thus retain production names and the structure of higher-order values, but do not keep track of the specific attribute instances that are accessed in each tree-creating expression. They abstract away local attributes, conditional expressions and specific attribute instance names, to generate a much simpler, albeit approximate model of the tree creation process. Note that while we generate rewrite rules for inherited accesses, these are meant to be place-holders. The rules do not model higher-order inherited attribute evaluation. This is handled separately as described in Sec. 5.

Fig. 6 shows a fragment of a higher-order attribute grammar, and the rules generated for its definitions. The left hand side of each rewrite rule is a simple production pattern consisting of the production name and a variable in each child slot. It rewrites tree (sub-) terms with that production at the root (the local's parent sub-tree) to the evaluated values of local tree terms. Thus the rules can be used to derive the value of each evaluated local from its parent's sub-tree. This means that for any tree creation sequence, we can construct a rewrite sequence of the same length. Thus if the rewrite rules terminate, then

no infinite tree creation sequence exists. We use existing tools such as AProVE that check term rewriting systems for termination to see if the generated rules terminate.

In any tree creation sequence, every non-initial tree can be derived via a non-empty rewrite sequence, from the sub-tree of its predecessor on which it was evaluated.

> Lemma 1: Given a tree creation sequence $t_0,\ t_1,\ t_2, ...$, if $n_i \in nodes(t_i)$ is the node on which tree t_{i+1} with root node r_{i+1} is evaluated, we have $term(n_i) \Longrightarrow^+ term(r_{i+1})$.

A proof for this theorem is given in [9]. The intuition behind it can be understood by referring to Fig. 1 and breaking each sequence into two parts, the first term, and the rest of the sequence generated from that first term by the rewrite rules. The first term in this generated rewrite sequence represents the tree value after all conditions in its defining expression have been evaluated. The rules model conditional expressions by generating multiple rules corresponding to the two clauses in the expression. The first term thus is a partially evaluated version of the expression in which all conditions have been evaluated, but in which attribute instance accesses are not computed and are represented by the sub-trees on which they are evaluated. This term is derivable from the tree term of the attribute instance's defining node, via the one rule that corresponds to the actual value of the conditional expression. In the first term in the sequence, attribute accesses off children or locals in the tree's defining expression are represented by the rewrite term of the child or local, respectively. Since the generated rules also model the evaluation of these attribute instances of the child or local tree terms, additional rewrites can be performed on these sub-terms to generate the actual evaluated attribute instance value. The fact that these additional rewrites can be performed to generate the correct value can be shown inductively on the structure of the defining expression. The inductive assumption is that all previously evaluated tree values have corresponding valid rewrite sequences from their parent sub-trees to the created values.

Given that we can construct sequences that derive each tree from its predecessor, we can construct a rewrite sequence that models the entire tree creation sequence. As each tree is evaluated on a node of its predecessor, the first term of the rewrite sequence that models this tree creation step is a sub-term of the predecessor tree. Thus the predecessor tree itself can be rewritten to a term in which the parent sub-tree is replaced by the local. This new term contains the new local as a sub-term, and therefore the process can be continued for the next evaluated local. As the rewrite sequence corresponding to each local's evaluation is non-empty, we can construct a rewrite sequence to model the entire tree creation sequence that is at least as long as the tree creation sequence. We can state this formally as the following theorem (proved in [9]).

Lemma 2: For a tree creation sequence $t_0,\ t_1,\ t_2, ...$, there is a function $R : N \longrightarrow Term$ so that for $i > 0,\ R(t_{i-1}) \Longrightarrow^{+} R(t_i)$.

Thus we have the following theorem for the case in which there are no higher-order inherited attributes.

Theorem 2: If the rewrite rules generated for a complete, non-circular grammar terminate, no improper evaluation sequence exists, in the absence of higher-order inherited attributes.

The proof is by contradiction. By Theorem 1, for any improper evaluation sequence for a tree t_0, there is an infinite tree creation sequence $t_0,\ t_1,\ t_2,\ ...$ By Lemma 2, we can define a function R so that for $i > 0,\ R(t_{i-1}) \Longrightarrow^{+} R(t_i)$. Thus there is an infinite rewrite sequence using the generated rules. Therefore the rewrite rules are non-terminating, which is a contradiction. Thus, if the rules terminate, there is no improper evaluation sequence.

5 Ordering Trees to Limit Inherited Accesses in Tree Sequences

The term rewriting rules described in the previous section cannot model tree creation in the presence of higher-order inherited attributes. Rules containing just production names and child variables cannot accurately model tree creation with inherited attributes. They do not incorporate the contextual information required to evaluate inherited attributes (attributes on the parent node). Since we can no longer derive each local from its parent's sub-tree, Lemma 1 no longer holds. Therefore, we can no longer construct a rewrite sequence corresponding to each tree creation sequence, and so Lemma 2 no longer holds.

We present an analysis that checks the grammar for restrictions that ensure that the number of inherited accesses in a tree creation sequence is bounded. We have shown above that rewrite rules model tree creation sub-sequences in which there are no inherited attributes. Thus if the rewrite rules terminate and the grammar satisfies the restrictions that ensure finite inherited accesses, then there are no infinite tree creation sequences and hence evaluation terminates.

The restrictions check if there is an ordering $\succeq$ on the non-terminals in the grammar that ensures a finite number of tree creation steps using higher-order inherited attributes.

Ordering Non-Terminals to Limit Inherited Attribute Accesses: Suppose there exists a well-founded, reflexive and transitive ordering $\succeq$ on the grammar's non-terminals that satisfies the conditions enumerated below. Fig. 7 gives a formal specification of the desired non-terminal ordering.

$\succeq$ is a relation on NT such that

1. $\succeq$ is reflexive.
2. $\succeq$ is transitive.
3. $\succ$ is well-founded.
4. $X \succeq Y$ if
   $(\exists p \in P \,.\, X = \texttt{lhs}(p),\ Y \in \texttt{rhs}(p)) \vee$
   $(\exists p \in P,\ l@p \,.\, X = \texttt{lhs}(p),\ Y = \texttt{type}_\texttt{a}(l)) \vee$
   $(\exists a_S \in A_S \,.\, a_S@X,\ Y = \texttt{type}_\texttt{a}(a_S))$.
5. $X \succ Y$ if $\exists a_I \in A_I \,.\, a_I@X,\ Y = \texttt{type}_\texttt{a}(a_I)$.

Fig. 7. An ordering on a grammar's non-terminals that ensures that the trees in any tree creation sequence are non-increasing.

1. Non-terminal symbols are non-increasing from the root node of a syntax tree to its leaves. In other words, if there is a production with X as its left-hand side and a right-hand side containing Y, then $X \succeq Y$.
2. The root non-terminal of any tree value is no larger than the node on which it was evaluated. Thus, if a local attribute of type Y occurs on a production with LHS X, then $X \succeq Y$. Similarly, if a synthesized attribute of non-terminal type Y occurs on X, then $X \succeq Y$.
3. Finally, if an inherited attribute of non-terminal type Y occurs on X, then X is larger than Y. That is, inherited occurs-on declarations are non-circular.

where

- $X \approx Y$ is the same as $X \succeq Y \wedge Y \succeq X$.
- $X \succ Y$ is the same as $X \succeq Y \wedge \neg\, X \approx Y$.
- $t_1 \succeq t_2$ is the same as $\mathit{symbol}(t_1) \succeq \mathit{symbol}(t_2)$. In other words, trees are compared using their root non-terminals.

Constructing the Desired Non-Terminal Ordering: Below we specify a sound and terminating procedure that, for a given attribute grammar, attempts to define an ordering on its non-terminals that satisfies the conditions in Fig. 7. Before doing so we introduce the following abbreviated notations to specify the conditions that the constructed ordering must satisfy.

- $S(X,\ Y) \equiv (\exists p \in P \,.\, X = \texttt{lhs}(p),\ Y \in \texttt{rhs}(p)) \vee$
  $(\exists p \in P,\ l@p \,.\, X = \texttt{lhs}(p),\ Y = \texttt{type}_\texttt{a}(l)) \vee$
  $(\exists a_S \in A_S \,.\, a_S@X,\ Y = \texttt{type}_\texttt{a}(a_S))$.
- $I(X,\ Y) \equiv \exists a_I \in A_I \,.\, a_I@X,\ Y = \texttt{type}_\texttt{a}(a_I)$.

We need a procedure that constructs an ordering $\succeq$ as required by Fig. 7 on a given attribute grammar's non-terminals so that for any $X, Y \in NT$, $S(X,\ Y)$

Procedure A:

1. Construct a directed graph G_{NT} whose vertexes correspond to the non-terminals in NT, and with an edge $\langle X,\ Y\rangle$ if and only if $S(X,\ Y)$.
2. Construct a directed graph G_{SCC} whose vertexes correspond to the strongly connected components (SCC) of G_{NT}, and with an edge $\langle S_X,\ S_Y\rangle$ if and only if there exist $X \in S_X$ and $Y \in S_Y$ where either $I(X,\ Y)$, or $S(X,\ Y)$ and $\neg S(Y,\ X)$.
3. If G_{SCC} has cycles, return failure, the required ordering doest not exist.
4. The desired ordering is defined as follows: $X \succeq Y$ if and only if there is a (possibly trivial) path in G_{SCC} from X's SCC to Y's SCC. Note that this means that $X \approx Y$ if and only if X and Y are in the same SCC in G_{NT}.

Fig. 8. A procedure that, for a given attribute grammar, attempts to define an ordering on its non-terminals that satisfies the conditions in Fig. 7.

implies $X \succeq Y$ and $I(X,\ Y)$ implies $X \succ Y$. Procedure A in Fig. 8 gives such a procedure.

The intuition behind Procedure A can be understood as follows. For any tree creation sequence, we can construct a path in G_{SCC} of the nodes that correspond to the strongly connected components (SCC) of the non-terminal symbols at the roots of the trees in the sequence. For "non-inherited" tree creation steps, we either stay at the same SCC node (in those cases in which $X \approx Y$), or move to another SCC node (in those cases in which $X \succ Y$). For all other tree creation steps, we move to another SCC node. Thus if the graph has no cycles, then there can only a finite number of "inherited" tree creation steps.

Lemma 3: The ordering $\succeq$ generated by Procedure A for a given attribute grammar satisfies the conditions in Fig. 7.

As an example, we consider an attribute grammar for a simple imperative language. Syntax trees for programs in this language have non-terminals that include statements (`Stmt`) which derive expressions (`Expr`) and types (`Type`). The symbol table for looking up variable names is implemented as a higher-order inherited attribute of type `Env` that decorates statements and expressions, as expected. Since `Env` occurs as an inherited attribute on the non-terminals appearing in the syntax tree, the algorithm in Fig. 8 computes the following:

`Stmt` $\approx$ `Expr`, `Stmt` $\approx$ `Type` and `Stmt` $\succ$ `Env` (and `Expr` $\succ$ `Env`, `Type` $\succ$ `Env`)

If such a language also allows names to be bound to types, as in the `typedef` construct in C, we might add a type environment (`TypeEnv`) that maps type

names to type representations (`TypeRep`). If this type environment was an inherited attribute on the environment (that would be used to look up type names that were bound to variables names in `Env`) then it would add another level to the ordering of non-terminals, resulting in the following:

$$\texttt{Env} \succ \texttt{TypeEnv} \text{ and } \texttt{Env} \succ \texttt{TypeRep} \text{ and } \texttt{TypeEnv} \approx \texttt{TypeRep}$$

Proving termination: If such an ordering exists, then every tree created as a local attribute is no larger (with respect to $\succeq$) than the tree on which it was created. Thus the trees in any tree creation sequence (corresponding to a path in the tree of locals) are non-increasing, as stated in the following lemma.

> Lemma 4: Assume $\succeq$ exists as described in Fig. 7. For any evaluation sequence with a tree creation sequence $t_0,\ t_1,\ t_2, ...$ we have $t_i\ \succeq\ t_{i+1}$, for all $i \geq 0$.

We define a *constant* tree creation sequence as a tree creation sequence in which there are no steps in which the generated tree is strictly smaller (with respect to $\succeq$) than its parent tree. It is thus a tree creation sequence $t_0, t_1, t_2, ...$ where $t_0 \approx t_1 \approx t_2 \approx ...$

> Lemma 5: If $\succeq$ exists as described in Fig. 7, then for any infinite tree creation sequence, there exists an infinite constant tree creation sequence.

Since the trees in a constant tree creation sub-sequence are not created using inherited attribute accesses, we can show that the rewrite rules defined in the previous section are sufficient to model such sub-sequences.

> Lemma 6: For a constant tree creation sequence $t_0,\ t_1,\ t_2,\ ...$, there is a function $R : N \longrightarrow \mathit{Term}$ so that for $i > 0,\ R(t_{i-1}) \Longrightarrow^{+} R(t_i)$.

The proof of this is similar to that for Lemma 2. The only additional complexity is that the rewritten terms are "pruned" versions of the actual tree terms, in which all sub-trees rooted at nodes less than the roots of trees in the constant tree creation sequence, are replaced by the `INH` place-holder term. In effect, we ignore the parts of the tree with non-terminals less than the roots of the trees in the constant tree creation sequence. This is necessary as the rewrite rules, lacking the context in which inherited attributes are evaluated, represent inherited accesses with the placeholder `INH`. Thus if higher-order inherited attributes are present, we can no longer generate the actual local trees using the rules. We can prove inductively that for constant tree creation steps, we can rewrite the pruned parent sub-tree tree to the pruned new local tree (that is not `INH`) via a non-empty rewrite sequence. The full proof is given in [9].

Thus if there exists an infinite constant tree creation sequence, then there exists an infinite rewrite sequence. So if the rules terminate, then all constant tree creation sequences terminate. Ordering non-terminals in this way therefore allows us to limit the number of inherited accesses in a tree creation sequence, and then use rewrite rules to model the parts of the sequence that do not use inherited attributes. We thus have the following theorem to show termination in the presence of higher-order inherited attributes.

> Theorem 3: If the rewrite rules generated for a complete, non-circular grammar terminate, and the non-terminals can be ordered as desired, no improper evaluation sequence exists.

The proof is by contradiction. By Theorem 1, for any improper evaluation sequence for a tree t_0, there is an infinite tree creation sequence $t_0,\ t_1,\ t_2,\ ....$ If the non-terminals can be ordered as required, then for an infinite tree creation sequence, there is an infinite constant tree creation sequence $t'_0,\ t'_1,\ t'_2,\ ...$, by Lemma 5. By Lemma 6, we can define a function R so that for $i > 0,\ R(t'_{i-1}) \Longrightarrow^{+} R(t'_i)$. Thus there is an infinite rewrite sequence using the generated rules. Therefore the rewrite rules are non-terminating, which is a contradiction. Thus if the rules terminate, and the non-terminals can be ordered as desired, there is no improper evaluation sequence.

Proofs of all theorems and their supporting lemmas are given in [9].

6 Discussion and Evaluation

The inspiration for using rewrite rules to model tree creation comes from a feature in Silver that can be rewritten using standard higher-order attribute grammars and thus is not part of the RHOAGs class. This feature is *forwarding* [13] and it has some similarities to rewriting in an attribute grammar setting. To use forwarding, a production in the grammar computes a new syntax tree and designates it as being "semantically equivalent" to the construct defined by the production. For example, the `doWhile` production in Fig 6 creates the local tree `fs` that is the translation of a do-while construct, as a sequence of the loop body and a while-loop with the same condition and body. This encodes the idea that "`doWhile(`c`) {`s`}`" is equivalent to "s`; while(`c`) {`s`}`". To use forwarding, this production could designate the `consStmt(s1, while(e, s1))` tree as its forwards-to tree. During attribute evaluation, if the "forwarding" production does not explicitly specify a definition for an attribute that is requested, then the production "forwards" that attribute query to the semantically equivalent forwards-to tree. The attribute is evaluated on that tree and returned to the original forwarding-tree, to be returned as the value of the attribute for the original query.

The concept of forwarding is useful in extensible language design [15], especially in avoiding defining attributes that can more easily be defined by translation to a "core" language. For example, an attribute used to translate this

sample language to machine code may be defined on the `while` production, but not on the `doWhile` production that forwards to the equivalent while-loop construct described above. In some sense, forwarding is non-destructive rewriting of the forwarding-tree to the forwarded-to-tree for attribute evaluation. Our initial interest was in showing that the process of forwarding would eventually terminate and forwarding's resemblance to rewriting led us in this direction. Forwarding can be translated away to higher-order attributes if extra "copy" rules are generated for each missing synthesized attribute equation that was handled by forwarding. For this analysis, it is possible to do this transformation without the loss of accuracy. Note that for other analyses, such as for completeness and circularity, translating away forwarding results in less precise analyses and thus this is not done for those analyses [13].

The problem of showing termination, whether for attribute evaluation or term rewriting systems, is undecidable, so our goal is an approximate solution. To some extent, it is similar to showing termination of the attribute evaluator program written in a programming language such as Haskell or Java. Silver translates attribute grammar specifications to Java and we could attempt to analyze that generated code. However analysing such lower-level implementations would discard much of higher-level domain-specific information about AGs that is useful in the analysis. We desire a simpler abstraction on which to perform analyses such as termination, one that incorporates our domain knowledge of AGs as well as information gained from our implementations of AG specifications for real-world programming languages. Our experience with grammars such as ableJ shows us that sticking to a few simple and reasonable guidelines regarding higher-order attribute definitions, will guarantee termination of higher-order attribution.

The design of the generated rewrite-rules that model tree creation is guided by several factors. Since a constantly failing analysis would technically be sufficient in a conservative analysis, the generated rules must be useful and be able to show termination of sophisticated grammars such as ableJ. Finally, we must be able to formally show that the rules are correct for our purposes; if the generated rules are terminating, then there are no infinite tree creation sequences and therefore attribution in general terminates. The nature of the problem requires us to walk a fine line between developing an analysis that is simple enough to be easily proven correct and efficiently executed, but is not so imprecise that it cannot show termination for a large class of useful and interesting grammars. In our experience with ableJ the rules we define satisfy both these conditions.

The termination analysis is conservative, in that there are terminating grammars for which the analysis fails. This follows first from the fact that both AProVE and the ordering generator are conservative. Further, the constructed rewrite rules are conservative in how they model tree creation since they derive more terms than are actually generated during higher-order evaluation. For example, attribute instance names are not retained in the sub-terms corresponding to synthesized attribute accesses. Thus the rules derive sequences corresponding to the evaluation of every possible higher-order synthesized instance on the child

or in-lined local. Only one of these will correspond to the actual run-time evaluation sequence in which a particular attribute instance is evaluated. Similarly, only one of the rules generated for the conditional expressions in a definition will model the tree created once all conditions are evaluated. While the rules are approximate in that they may not be able to show termination for all terminating grammars, they are correct. They can show termination for non-trivial grammars such as ableJ in a reasonable amount of time.

While the class RHOAG is expressive enough to include grammars similar to ableJ, there are some grammars and Silver features that are beyond the scope of our analysis. Firstly, it does not handle certain kinds of attribute definitions in which the generated tree has the same production name as the defining production and the child trees are retrieved via accesses to synthesized attributes on child nodes. An example would be a synthesized attribute that stores the "base" versions of "extended" statements such as the `doWhile` and `ifThen` productions in Fig. 6. For "base" productions such as `while` and `ifThenElse` this attribute would be constructed with the same production. Unfortunately the rules we generate for such definitions are always non-terminating and therefore not useful. Finally, since the analysis assumes that the grammar is non-circular, it does not handle reference attribute grammars since the circularity problem is undecidable in that setting.

7 Related Work

In Knuth's original work on attribute grammars [8] he gave a circularity analysis. In the attribute grammar community it is not uncommon to extend this analysis when new features are added to the attribute grammar paradigm. In their specification of higher-order attribute grammars, Vogt *et al.* extended Knuth's circularity analysis to that setting. In the original work on *forwarding* the completeness analysis was combined with the circularity analysis to accommodate the need to use global attribute dependency information in the completeness analysis [13, 1]. In the case of reference and remote attributes [6, 2], it is not possible to have a precise circularity analysis, as the problem is undecidable [2].

In Vogt *et al.*'s original paper on HOAGs they define "well-definedness" to include termination of tree construction and state a lemma [17, Lemma 3.2, page 142] that imposes a condition that would ensure termination of tree construction. This condition requires that non-terminal attributes (the terminology used for higher-order attributes) do not appear more than once in a path in their formulation of trees. This is similar to our imposition of an ordering on non-terminals for higher-order inherited attributes. But our use of rewrite rules for local and synthesized higher order attributes is less restricting and allows for a more precise analysis. In other work [16] they drop this termination requirement for grammars to be considered "well-defined".

There is also a trove of related work in the area of program termination for functional and imperative programming and by necessity we cannot attempt to cover this area. One track of work began with the work by Lee, Jones and

Ben-Amram[10] on the "size change" principle applied to first order functional programs. Here, the *size* of a parameter for each function call must always decrease over a domain with a fixed smallest value. This was later extended to higher-order functional programs [11]. This work was done for call-by-value languages. While some techniques in these works might also be useful in an analysis on higher order attribute grammar termination, they are not immediately applicable since they are designed for imperative or functional programming languages. We could, in principle, translate the attribute grammar specification to a functional program [7] (in fact, Silver formerly translated attribute grammar specifications to Haskell). But as described above, an analysis that uses the domain knowledge captured in the attribute grammar constructs is simpler and the use of such domain knowledge often leads to more precise analyses.

8 Future Work and Conclusion

There are several possible avenues for future work. First, we will look at relaxing the restrictions on the class RHOAG of higher-order attribute grammars handled by our termination analysis. We would like to handle grammars with attribute definitions that construct trees with the same production name as the attribute's production. We would also like to handle grammars that make use of Silver's `case` constructs. Finally we would like to relax the analysis' and its correctness proof's assumptions of grammar non-circularity in order to handle reference attribute grammars (RAGs).

The rules we generate are simple and do not make full use of the analytical power of tools such as AProVE. We expect that much simpler analyses could be used to show termination for the rules generated for most grammars. One possible approach would be to order productions (based on tree creating definitions) and then verify that the generated rules satisfy the corresponding recursive path ordering (RPO) [3]. This would provide a possible avenue for modularizing the analysis. We expect to be able to specify a modular condition (on extension specifications) that ensures the existence of the desired production ordering in the composed grammar. We also expect to be able to specify a modular condition on extensions that guarantees that the composed grammar's non-terminals can be ordered to limit the number of inherited accesses in tree creation sequences. Such modular analyses are important in our Silver extensible language framework, which aims at the development of modular and composable language extensions.

To conclude, in this paper we described an analysis that checks attribute grammars for termination of higher-order attribute evaluation. With the higher-order circularity and completeness tests, this analysis ensures that attribute evaluation terminates normally. We have implemented this analysis in Silver and have run this analysis on ableJ, an extensible specification of the Java 1.4 programming language. Future work will include enlarging the class of HOAGs handled by the termination analysis, and developing a modular version for use in our extensible language framework.

References

1. Backhouse, K.: A functional semantics of attribute grammars. In: 7th Conf. on Tools and Algorithms for the Construction and Analysis of Systems, (TACAS 2001). LNCS, vol. 2280, pp. 142–157. Springer-Verlag (2002)
2. Boyland, J.T.: Remote attribute grammars. J. ACM 52(4), 627–687 (2005)
3. Dershowitz, N.: Orderings for term-rewriting systems. Theoretical Computer Science 17, 279–301 (1982)
4. Ekman, T., Hedin, G.: The JastAdd system - modular extensible compiler construction. Science of Computer Programming 69, 14–26 (December 2007)
5. Giesl, J., Thiemann, R., Schneider-kamp, P., Falke, S.: AProVE: A system for proving termination. In: Extended Abstracts of the 6th International Workshop on Termination, WST03. pp. 68–70 (2003)
6. Hedin, G.: Reference attribute grammars. Informatica 24(3), 301–317 (2000)
7. Johnsson, T.: Attribute grammars as a functional programming paradigm. In: Proc. of Functional Programming Languages and Computer Architecture. LNCS, vol. 274, pp. 154–173. Springer-Verlag (1987)
8. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2(2), 127–145 (1968), corrections in **5**(1971) pp. 95–96
9. Krishnan, L.: Composable Semantics Using Higher-Order Attribute Grammars. Ph.D. thesis, University of Minnesota, Department of Computer Science and Engineering, Minneapolis, Minnesota, USA (2012), to appear, draft available at `http://melt.cs.umn.edu/pubs/krishnan2012PhD/`
10. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Proc. of the 28th ACM Symposium on Principles of Programming Languages (POPL 2001). pp. 81–92. ACM Press (2001)
11. Sereni, D., Jones, N.D.: Termination analysis of higher-order functional programs. In: Proc. of the Third Asian Symposium on Programming Languages and Systems (APLAS 2005). LNCS, vol. 3780, pp. 281–297. Springer-Verlag (2005)
12. Sloane, A.M.: Lightweight language processing in Kiama. In: Proc. of the 3rd Summer School on Generative and Transformational Techniques in Software Engineering III (GTTSE 2009). LNCS, vol. 6491, pp. 408–425. Springer (2011)
13. Van Wyk, E., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in attribute grammars for modular language design. In: 11th Conf. on Compiler Construction (CC 2002). LNCS, vol. 2304, pp. 128–142. Springer-Verlag (2002)
14. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. Science of Computer Programming 75(1–2), 39–54 (January 2010)
15. Van Wyk, E., Krishnan, L., Schwerdfeger, A., Bodin, D.: Attribute grammar-based language extensions for Java. In: European Conf. on Object Oriented Prog. (ECOOP 2007). LNCS, vol. 4609, pp. 575–599. Springer-Verlag (2007)
16. Vogt, H.: Higher order attribute grammars. Ph.D. thesis, Department of Computer Science, Utrecht University, The Netherlands (1989)
17. Vogt, H., Swierstra, S.D., Kuiper, M.F.: Higher-order attribute grammars. In: ACM Conf. on Prog. Lang. Design and Implementation (PLDI 1989). pp. 131–145 (1989)