Acta Informatica 24, 633–651 (1987)

Acta Informatica

Lifetime Analysis for Attributes

Uwe Kastens
Universität-GH Paderborn, Fachbereich Mathematik-Informatik, D-4790 Paderborn, Federal Republic of Germany

Summary. Reduction of attribute storage is a vital requirement for attribute evaluators. We present a new method for the analysis of lifetime of attribute instances. It is assumed that attribute evaluation is performed by a visit-oriented evaluator. Its evaluation sequence for any input can be described by a contex-free grammar derived from the visit-sequences. Conditions on that CFG decide for each attribute whether all its instances can be stored in a single global variable. Furthermore one can decide whether several different attributes can be mapped to a single global variable. Similarly conditions for stack implementation are given. All decisions can be made efficiently at evaluator generation time. Hence the method is well suited for compiler generation.

1. Introduction

Any practical attribute evaluator has to take special care of storage used for attribute values: In a naive implementation of an attributed structure tree the tree nodes are records containing a component for each attribute instance. That is not tolerable in practice because: On the one hand the number of tree nodes and attributes is usually large, on the other hand the time between computation and last use of an attribute value, i.e. its lifetime, is often short. Hence one tries to eliminate the storage for certain attributes from the tree and allocate them as global variables or stacks instead. It must be ensured that these global objects contain the correct attribute value at any time. On the base of lifetime analysis one can decide for an attribute evaluator which attributes can be stored globally and hence reduce its runtime storage requirements.

Different ways lead to a solution of this problem:

a) Consider a particular tree for a certain input and decide globalization for the attribute instances of that tree. In this case decision analysis and the consequences thereof are performed at attribute evaluator runtime.
b) Consider an attribute evaluator for an attribute grammar and decide

globalization such that it is applicable for any tree. In this case decision analysis is performed at generation time resulting in a modified attribute evaluator (resp. its control tables)

c) Consider an attribute grammar and a given set of attributes to be globalized. Then construct an attribute evaluator with an evaluation sequence control which guarantees that the global variables are used properly – if such an evaluation sequence exists.

From the view of generation of efficient attribute evaluators we prefer method (b) because runtime is not burdened with additional analysis. In most cases runtime even decreases. Method (c) is less automatical because it requires a storage mapping given by the designer.

In this paper we first introduce a method for analysis of lifetime of attribute instances. We assume that a tree-walking attribute evaluator is given by a set of visit-sequences. For a large class of attribute grammars (simple-multi-visit AGs, ordered AGs, and all pass-oriented AGs, cf. [1, 2]) they can be computed from the attribute dependencies. The visit-sequences are transformed into a context-free grammar. Its sentences describe the sequence of attribute accesses the evaluator performs for any input. For this language one can decide whether the lifetimes of certain attribute instances are disjoint and in consequence such an attribute can be implemented by a global variable. Furthermore attributes can be mapped to global stacks if certain criteria on the visit-sequences hold.

In contrast to the approach of Ganzinger [3] (which follows method (b) too) we first decide for each attribute whether it can be implemented globally. Then we map different global attributes onto single variables. As a side-effect identical attribute assignments can be eliminated. We give an efficient algorithm for the globalization decision whereas the optimal solution of all three problems together (globalization, grouping, and identical assignment elimination) is shown to be NP-complete in [3].

Räihä presents in [4] a different method which requires a certain amount of lifetime analysis to be done at runtime (combination of methods (a) and (b)). For pass-oriented AGs techniques are wellknown which chose stack implementation for those attributes used in one pass only (method (b), [5, 6]). Our stack conditions are applicable for a larger AG class. They yield the same results in the special case of pass-oriented evaluation.

Knowing that there is some freedom in the construction of visit-sequences one can apply certain strategies which improve the results of lifetime analysis, e.g. "lazy evaluation" as described in [5] or for OAGs in [2] which shortens the lifetimes and reduces the number of overlaps. In [7] the opposite direction is taken by method (c): An intended mapping of attributes to global variables is given. Certain sufficient conditions on the attribute dependencies decide whether an evaluation sequence exists for any input, which then is determined at runtime.

The method presented here resulted from reasoning about the lifetime analysis in the compiler generating system GAG [8]. The technique used there is based on the same principle (analysis of visit-sequences), but it is less systematical and the globalization conditions are more pessimistic. (See [6] for its comparison with that used in a pass-oriented attribute evaluator.) In spite of that the practical

results with the GAG system are very encouraging for the planned substitution of the GAG optimization. (See [8] for measurements.)

In the following we assume that the reader has a basic understanding of attribute grammars. The notions used here are close to those of [2]. In Sect. 2 we introduce the underlying attribute evaluator and visit-sequences. They are transformed in Sect. 3 in order to describe attribute lifetimes and to formulate conditions for implementation of attributes by global variables. Sect. 4 gives weaker conditions for mapping attributes to global stacks. Throughout the text we use a running example for explanation. The results for a more realistic but still artificial example are given in the appendix.

2. Visit-Oriented Attribute Evaluators

In this section we introduce the preliminaries for our method of attribute lifetime analysis. We omit an introduction and a formal definition of attribute grammars and a discussion of attribute evaluator construction. The reader is referred to the literature on these topics. Our terminology and notation is close to that used in [2] and [8]. Our notation is introduced by an example AG which we will refer to subsequently. We briefly explain the principle of attribute evaluators controlled by visit-sequences which are presented in [2]. Finally a context-free grammar is derived from the visit-sequences describing operation sequences of the evaluator.

We introduce our terminology for AGs by reference to a small artificial AG in Fig. 1. It is a "classical" AG which can be evaluated by visit-oriented but not by pass-oriented evaluators. References to that example are attached in parenthesis.

An AG is based on a context-free grammar G (in the example given by the productions $p1$, $p2$, $p3$). It is augmented by a set of attributes A which is the disjoint union of the attribute sets A_X associated to each symbol X of the vocabulary of G ($A=A_Y=\{a, b, c, d\}$, $A_S=\emptyset$). The attributes of different symbols are considered to be different elements of A regardless whether they are named identically. If we want to stress that $a\in A_X$ we write $X.a$. Each A_X is subdivided into two disjoint subsets AS_X and AI_X of synthesized and inherited attributes ($AS_Y=\{b, d\}$, $AI_Y=\{a, c\}$). A set of attribute rules associated with each production of G defines the computation of attribute values.

The X_i in a production $p: X_0 ::= X_1 \dots X_n$ are occurrences of symbols of G. Correspondingly $A_p=\bigcup_{i=0}^{n} A_{X_i}$ are the attribute occurrences of p. We distinguish between defining occurrences $Ad_p=AS_{X_0}\cup\bigcup_{i=1}^{n} AI_{X_i}$ and applied occurrences $Aa_p=AI_{X_0}\cup\bigcup_{i=1}^{n} AS_{X_i}$. For each $a\in Ad_p$ there is an attribute rule associated to p defining a by a function depending on some attributes of Aa_p (all to a32 in Fig. 1). For ease of presentation we assume the AG to be in Bochmann Normal Form (i.e. the functions must not depend on Ad_p). All conditions could

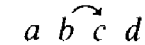
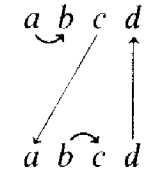
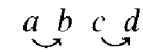
Production		Attribution		Dependencies
(p1) $S::=Y$	(a11)	$Y.a:=1$	S	
	(a12)	$Y.c:=f_1(Y.b)$	Y	
(p2) $Y::=xY$	(a21)	$Y_1.b:=f_2(Y_1.a)$	Y	
	(a22)	$Y_2.a:=f_3(Y_1.c)$		
	(a23)	$Y_2.c:=f_4(Y_2.b)$		
	(a24)	$Y_1.d:=f_5(Y_2.d)$	xY	
(p3) $Y::=z$	(a31)	$Y.b:=f_6(Y.a)$	Y	
	(a32)	$Y.d:=f_7(Y.c)$	z	

Fig. 1. An AG example with dependency graphs

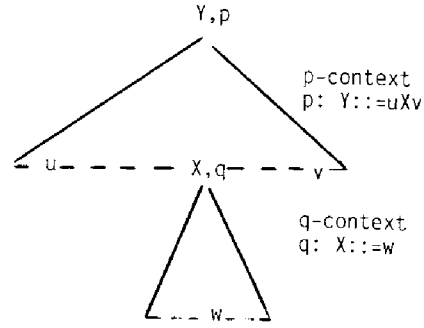


Fig. 2. Adjecent p- and q-context

be rewritten such that the functions may depend on attributes of A_p. The dependencies between attribute occurrences of p are shown in dependency graphs as in Fig. 1.

For our lifetime analysis we assume that attribute evaluation is performed by visit-oriented evaluators based on the following principles: Let t be a structure tree for a sentence of $L(G)$. Each node is labelled by a symbol of G and the production applied for its derivation (see Fig. 2). For each attribute of A_X an attribute instance is associated with each node labelled with X. The attribute evaluator computes a value for all attribute instances in the tree. For that purpose the evaluator walks through the tree controlled by some strategy. Visiting a node labelled with production p it may compute the values of some attribute instances corresponding to Ad_p applying attribute rules associated to p. We say the attribute evaluator is in a p-context. That means in Fig. 2 instances of inherited attributes of X are computed in p-context while those of synthesized attributes are computed in q-context. Hence such an evaluator performs three kinds of operations at some node k representing X_0 in a p-context, where $p: X_0::=X_1 \cdots X_n$:

$f(a_1, \ldots, a_j) \to X_i.a$	evaluation of an attribute rule associated to p, $i \in \{0, \ldots, n\}$
$\downarrow_j X_i$	j-th visit to the i-th descendant node of k
$\uparrow_j$	j-th visit to the ancestor node of k,

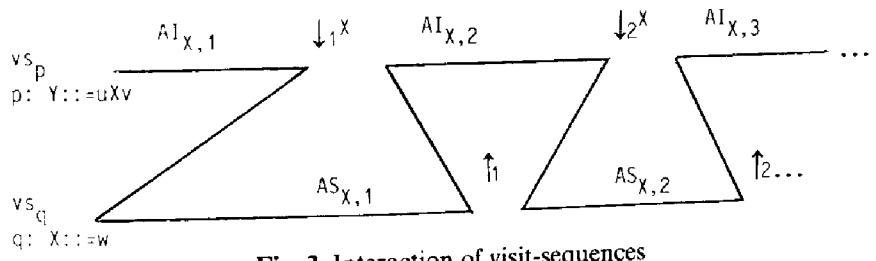


Fig. 3. Interaction of visit-sequences

$$vs_{p1}: \quad 1 \to Y.a, \downarrow_1 Y, f_1(Y.b) \to Y.c, \downarrow_2 Y, \uparrow_1$$

$$vs_{p2}: \quad f_2(Y_1.a) \to Y_1.b, \uparrow_1,$$
$$f_3(Y_1.c) \to Y_2.a, \downarrow_1 Y_2, f_4(Y_2.b) \to Y_2.c, \downarrow_2 Y_2, f_5(Y_2.d) \to Y_1.d, \uparrow_2$$

$$vs_{p3}: \quad f_6(Y.a) \to Y.b, \uparrow_1, f_7(Y.c) \to Y.d, \uparrow_2$$

Fig. 4. Visit-sequences for example AG

For a large subclass of well-defined AGs (simple-multi-visit AGs including OAGs and all pass-oriented AGs) such an evaluator can be controlled by a set of visit-sequences VS, one for each production p, each consisting of a sequence of the above operations. The visit-sequences are fixed at evaluator generation time on the basis of the attribute dependencies specified in the AG. They are applicable to the structure tree of any sentence in $L(G)$. An evaluator controlled by visit-sequences performs a proper tree walk starting at the root of the tree moving along its edges, evaluating all attribute instances, and finally returning to the root

Typical for this class of evaluators is the interleaved execution of visit-sequences vs_p and vs_q for adjacent contexts as shown in Fig. 3. The node for X with its attributes acts as an interface between evaluation of its subtree and the rest of the tree. The visit-sequences for our running example are given in Fig. 4.

For the construction of visit-sequences we refer to [2] and [9]. Here only the underlying principle is explained. For all nodes of any attributed tree which are labelled with the same symbol X the interface behaviour is identical, independent of its context. Such an interface is described by a linear order over subsets of the attributes A_X: Let A_X be partitioned into disjoint subsets $AI_{X,i}$ and $AS_{X,i}$ for $i=1, \ldots, m_X$. (For ease of description the indices i used here differ from those in [2] the correspondence should be obvious.) Then the inherited attributes in $AI_{X,i}$ are evaluated before the i-th visit to a node labelled with X. The synthesized attributes in $AS_{X,i}$ are evaluated during the i-th visit. Hence this linear order over the attribute subsets defines a partial order over A_X. In Fig. 3 the subsets of A_X mark the sections of the visit-sequences where they are computed.

For simple multi-visit AGs [1], OAGs [2], and all pass-oriented AGs there is a partition of all A_X with the following property: In any attributed tree

the graph of the dependencies between the attribute instances overlayed by those partial orders is acyclic. Given such partitions for all attribute sets it is easy to compute a visit-sequence vs_p for each production p from the attribute dependency graph associated to p. An efficient algorithm for partition computation in the case of OAGs is given in [2]. In the case of pass-oriented AGs that computation is trivial. The partition for A_Y of our example is

$$AI_{Y,1}=\{a\} \quad AI_{Y,2}=\{c\}$$
$$AS_{Y,1}=\{b\} \quad AS_{Y,2}=\{d\}.$$

It is a property of this particular example that each partition contains exactly one attribute. Visit-sequences constructed by the above method ensure that for an attribute computation all arguments are available. Each attribute a is available before (after) the j-th visit to its symbol node if $a \in AI_{X,j} (a \in AS_{X,j})$. In some contexts it may be possible to evaluate a earlier; but due to the interface concept no advantage would be taken from that – only its lifetime would be lengthened. Hence we assume that each attribute is computed exactly in those parts of visit-sequences determined by the attribute partitions.

3. Lifetime Analysis

In the following we analyze lifetimes of the attributes. So we transform the visit-sequences such that they describe only begin and end of lifetime, i.e. definition and last use of attributes. The result is a new set of lifetime visit-sequences denoted by $vsl_p \in VSL$.

We obtain the vsl_p from vs_p by three transformation steps:

1. The function and constant symbols are dropped from vs_p. Instead of $f(a, b, c) \to d$ we get $abc \to d$ in vsl_p, meaning that a, b, and c are used (for some computation not of interest here) and then d is defined.
2. For each attribute of Aa_p all applications except the last one are eliminated, in order to mark only the end of its life time.
3. It may be that in certain vs_p an attribute $a \in Aa_p$ is not applied at all (consider d in vs_{p1} of our example). In order to close its life time properly we introduce in vsl_p an artificial last application: If $a \in AS_{X,j}$, a is computed in the j-th visit to its symbol X. Hence we insert that last application immediately after that visit $\downarrow_j X$ – the earliest possible point in vsl_p. If $a \in AI_{Xj}$, a is computed before the j-th visit to its symbol $X = X_0$. In vsl_p that visit reaches either the beginning of vsl_p (if $j=1$) or the point following the $(j-1)$-st ancestor visit. That is the earliest point where the artificial last application can be inserted.

In our example each attribute of Aa_p is used exactly once, except d which is not used in vs_{p1}. Hence an application of d is inserted in vsl_{p1}. Fig. 5 shows the vsl_p for our example.

An interpreter which is controlled by the so constructed lifetime visit-sequences performs exactly the same walk through a given structure tree as an attribute evaluator controlled by the *VS* does. Instead of computing attribute

$$vsl_{p1}: \rightarrow Y.a, \downarrow_1 Y, Y.b \rightarrow Y.c, \downarrow_2 Y, Y.d, \uparrow_1$$
$$vsl_{p2}: Y_1.a \rightarrow Y_1.b, \uparrow_1, Y_1.c \rightarrow Y_2.a, \downarrow_1 Y_2, Y_2.b \rightarrow Y_2.c, \downarrow_2 Y_2, Y_2.d \rightarrow Y_1.d, \uparrow_2$$
$$vsl_{p3}: Y.a \rightarrow Y.b, \uparrow_1, Y.c \rightarrow Y.d, \uparrow_2$$

Fig. 5. Lifetime visit-sequences for example AG

values this interpreter begins and ends lifetimes of attribute instances interpreting a defining or the last applied occurrence respectively. One can imagine that it allocates and frees storage for them. For each given structure tree it performs a certain sequence of these operations.

We do not suggest to run such an interpreter. But we want to study properties of the complete set of operation sequences for all structure trees. Especially the following question shall be answered: Can all instances of an attribute (or the attributes of a set $B \subseteq A$) be implemented by a single global variable such that the correct value is available at any time. So for the operations of the interpreter we do not distinguish different instances or different occurrences of an attribute a. Hence the operations (apart from the tree moves) are

D_a defining an instance of a, i.e. begin of its lifetime,
L_a last use of an instance of a, i.e. end of its lifetime.

For our example one can easily deduce from the attribute dependencies that for a tree of height $n+1$ the operation sequence is

$$(D_a L_a D_b L_b D_c L_c)^n (D_d L_d)^n.$$

Assume that an evaluator is given by a set VSL. For each tree representing a certain input to the evaluator there is a certain operation sequence. Let SEQ be the set of all operation sequences for any tree defined by the underlying context-free grammar. SEQ is a language over an alphabet

$$T_L = \{D_a | a \in A\} \cup \{L_a | a \in A\}.$$

In order to state the global variable condition for a certain set B of attributes we define a projection of SEQ describing only the lifetimes of instances of attributes in B:

Let SEQ be the set of operation sequences of an interpreter controlled by lifetime visit-sequences VSL, and $B \subseteq A$ a set of attributes. Then $proj(s, B) = sb$ projects $s \in SEQ$ to $sb \in \{D, L\}^*$ such that each element D_a, L_a of s is mapped to D or L in sb respectively if $a \in B$, or to the empty string if $a \notin B$. With the mapping $proj$ we get

$$SEQ^B = \{sb | sb = proj(s, B), s \in SEQ\} \subseteq \{D, L\}^*.$$

SEQ^B can be understood as the set of operation sequences for a modified interpreter which takes care of only the instances of attributes in B, and which

does not distinguish between instances of different attributes in B. Some projections for our example are

$$SEQ^B=(DL)^n \quad \text{for } B=\{a\}, \quad B=\{b\}, \quad B=\{c\}, \quad \text{or } B=\{d\},$$
$$SEQ^B=(DL)^{4n} \quad \text{for } B=\{a, b, c, d\}.$$

The attributes in B can be implemented by a single global variable if in all sequences in SEQ^B the lifetimes are disjoint.

Theorem 1. *Let $B\subseteq A$ and SEQ^B be the projection of the operation sequences for an interpreter controlled by a given set of lifetime visit-sequences. Then all attributes in B can be implemented by a single global variable if and only if $SEQ^B\subseteq(DL)^*$.*

Proof. The construction of *VSL* guarantees that a D-L-pair for any attribute instance of B is contained in a sentence of SEQ^B. If $SEQ^B\subseteq(DL)^*$ holds, then each L symbol refers to the same attribute instance as its immediate predecessor D does. Hence the lifetime of all attribute instances for B are disjoint.

On the other hand assume that all attributes in B can be implemented by a global variable. Then the begin of a lifetime (D) must be followed by its end (L) before the lifetime for another instance is opened. Hence $SEQ^B\subseteq(DL)^*$ must hold. □

In order to compute *SEQ* formally and to check the property of Theorem 1 efficiently the set of lifetime visit-sequences *VSL* is transformed into a context-free grammar G_L such that $L(G_L)$ contains *SEQ*. The basic idea of the transformation refers to the interleaved execution of visit-sequences (Fig. 3): A descendent visit $\downarrow_j X$ causes execution of a section of a visit-sequence ending with an ancestor visit $\uparrow_j$ associated to a production with X on its lefthand side. All these sections become productions of G_L and the descendant visits are transformed into occurrences of suitable nonterminals of G_L.

The transformation of the lifetime visit-sequences into a lifetime grammar $G_L=(N_L, T_L, P_L, S_L)$ is defined by the following rules:

a) For each nonterminal X in G there are k nonterminals $X^1, \ldots, X^k$ in G_L if X is visited $k=m_x$ times. $N_L=\{X^j \mid X\in N, j=1, \ldots, m_X\}$
b) $S_L=S^1$ (The start symbol is visited only once.)
c) Each applied (defined) attribute occurrence of an attribute $a\in A$ is transformed into a terminal $L_a(D_a)$
d) In vsl_p associated to a production p: $X_0::=X_1 \ldots X_n$ each visit $\downarrow_j X_i$ is transformed into an occurrence of X_i^j.
e) Let $vsl_p=u_1\uparrow_1 u_2\uparrow_2 \ldots u_k\uparrow_k$ be associated to a production of G as in (*d*). Then it is transformed into k productions of G_L.

$$X_0^1::=v_1 \quad X_0^2::=v_2 \ldots X_0^k::=v_k$$

Where the v_i are the transformations of the u_i, according to (c) and (d).

The grammar G_L for our running example is given in Fig. 6. In this case one easily verifies that $L(G_L)=(D_a L_a D_b L_b D_c L_c)^a(D_d L_d)^n$ as stated above.

Lemma 1. *Let G_L be constructed from a set of lifetime visit-sequences VSL, and let SEQ be the set of operation sequences of an interpreter controlled by the same VSL. Then $SEQ\subseteq L(G_L)$ holds.*

$$S^1 ::= D_a\, Y^1\, L_b\, D_c\, Y^2\, L_d$$
$$Y^1 ::= L_a\, D_b$$
$$Y^2 ::= L_c\, D_a\, Y^1\, L_b\, D_c\, Y^2\, L_d\, D_d$$
$$Y^2 ::= L_c\, D_d$$
$$L(G_L) = (D_a\, L_a\, D_b\, L_b\, D_c\, L_c)^n (D_d\, L_d)^n$$

Fig. 6. Lifetime grammar G_L for the example AG

Proof. Assume that $s \in SEQ$ is the interpreter sequence obtained for some tree t. Then there is a derivation for G_L: $S^1 \overset{*}{\Rightarrow} u X^j w \Rightarrow uvw \overset{*}{\Rightarrow} s$ such that X^j corresponds to the j-th descendent visit to a node labelled with X and derived by a production p. v describes the operations of the interpreter performed for that visit to that node controlled by the section of vls_p which ends with $\uparrow_j$. Hence that derivation exactly reflects the tree walk of the interpreter. □

The opposite direction $L(G_L) \subseteq SEQ$ does not hold in general. Hence lifetime analysis based on G_L can be pessimistic in cases where $SEQ \neq L(G_L)$ which are characterized in Lemma 2 below.

The result of Lemma 1 allows to efficiently check global variable conditions for SEQ by checking them for $L(G_L)$. In order to prove the condition for a certain set $B \subseteq A$ of attributes we project G_L to G_L^B by a mapping which corresponds to the projection of SEQ to SEQ^B:

For a given set $B \subseteq A$ $G_L^B = (T^B, N_L, P_L^B, S_L)$, $T^B = \{D, L\}$, and $S_L = S^1$ the productions P_L^B are constructed by replacing in the right hand side of each production of P_L the symbols D_a, L_a by D or L respectively if $a \in B$, and by the empty string if $a \notin B$.

From Lemma 1 immediately follows that $SEQ^B \subseteq L(G_L^B)$. Hence we have to check $L(G_L^B) \subseteq (D_L)^*$. The problem whether the language of a context-free grammar is contained in a given regular language is decidable. In this special case that property can be checked by a simple and efficient algorithm using the FIRST and FOLLOW sets, well-known from grammar analysis. The conditions

a) FIRST $(S_L) = \{d\}$,

b) FOLLOW $(D) = \{L\}$, and

c) FOLLOW $(L) = \{D, \varepsilon\}$

are equivalent to $L(G_L^B) \subseteq (DL)^*$. Furthermore condition (a) is implied by the consistency of the visit sequences.

For practical implementation in a system which generates attribute evaluators automatically (like GAG) one would proceed in two steps: First for each attribute $a \in A$ a singleton set $B = \{a\}$ is considered, and it is decided whether it can be implemented by a global variable using the above algorithm. The result is a set $GI \subseteq A$ of global attributes. In the second step subsets of GI are considered as groups of attributes for one global variable. For them the condition is checked again. (Of course no attribute not in GI can contribute to such a group.) The grouping can either be given by design decisions of the user which are checked by the system, or it can be computed by an algorithm

of the system. (The GAG system comprises both facilities, but based on a more pessimistic global condition than that presented here.) For the latter case we got good results using a first-fit algorithm [8]. As an additional effect one can eliminate all identical attribute assigns between attributes of B.

It should be stressed that the more significant storage improvement is achieved by the first step. Whereas the effect of grouping is rather low, because only the number of global variables is reduced compared with the elimination of many attribute instances from the tree. Hence we do not recommend an algorithm which computes a storage optimal grouping in B. The more significant effect of grouping is achieved by elimination of identical attribute assigns between attributes of single groups in B. That is an even more complex optimization problem (cf. [3]) which in practice will be solved by efficient but suboptimal algorithms.

Now we have a closer look at the difference between SEQ^B and $L(G_L^B)$, which causes the global condition to be pessimistic in some cases: If an interpreter of the lifetime visit-sequences visits a node more than once it will always resume the same visit-sequence. On the other hand in a derivation according to G_L^B at the corresponding places productions transformed from different visit-sequences may be applied. That results in sentences $s \in L(G_L^B)$ which are not in SEQ^B.

Lemma 2. *Each of the following conditions implies that* $SEQ = L(G_L)$ *holds:*

a) *If* P_L *of* G_L *does not contain four productions of the form*

$$X^i ::= s, \quad X^j ::= t \quad \text{result of the transformation of } vsl_p$$
$$X^i ::= s', \quad X^j ::= t' \quad \text{result of the transformation of } vsl_{p'}$$

where $i \neq j$, $s \neq s'$, *and* $t \neq t'$. p *and* p' *have the same left hand side.*

b) *If for each symbol* $X \in N$ *which is visited more than once there is exactly one production in* P.

c) *The AG is a simple-one-visit AG (or a subclass thereof, e.g. LAG(1), AAG(1)), and the (canonical) visit-sequences are chosen for VSL resulting from the check of the grammar class.*

Proof. For (*a*) the proof is obvious since a situation as described above can only arise from a derivation in G_L of the form

$$S_1 \overset{*}{\Rightarrow} x Y^k y \Rightarrow x u X^i v X^j w y \Rightarrow x u s v t' w y \Rightarrow \ldots,$$

It applies productions of different visit-sequences in the same context. A lifetime interpreter controlled by *VSL* however applies sections of the same visit-sequence at one tree node. Hence the derived terminal string $z \notin SEQ$. Furthermore conditions (c) implies (b), and (b) implies (a). □

In some of the cases where $L(G_L^B)$ is pessimistic compared to SEQ^B we can deduce from G_L^B that the global variable condition fails due to sentences not in SEQ^B. We can eliminate some of these sentences (not all) by lengthening the lifetime of attributes in certain contexts. Consider the following subsets of productions in some G_L^B (productions in the same line are derived from the

same vsl_p. We assume that u, v, w derive to $(DL)^*$, i.e. they do not violate the global condition.

a)

$$\begin{array}{lll} Z^1 ::= DX^1\, u X^2 & & (vsl_{p1}) \\ X^1 ::= v & X^2 ::= L & (vsl_{p2}) \\ X^1 ::= L & X^2 ::= w & (vsl_{p3}) \end{array}$$

Z_1 derives to $DvuL$ and $DLuw$ but also to $DLuL$ and $Dvuw$

b)

$$\begin{array}{lll} Z^1 ::= X^1\, u X^2 & & (vsl_{p1}) \\ X^1 ::= DY^1 & X^2 ::= Y^2 & (vsl_{p2}) \\ X^1 ::= v & X^2 ::= DY^1\, w\, Y^2 & (vsl_{p3}) \\ Y^1 ::= s & Y^2 ::= L & (vsl_{p4}) \end{array}$$

Z^1 derives to $DsuL$ and $vuDswL$ but also to $DsuDswL$ and vuL

c)

$$\begin{array}{lll} Z^1 ::= X^1\, u X^2 & & (vsl_{p1}) \\ X^1 ::= Y^1 L & X^2 ::= v & (vsl_{p2}) \\ X^1 ::= w\, Y^1 & X^2 ::= L & (vsl_{p3}) \\ Y^1 ::= D & & (vsl_{p4}) \end{array}$$

Z^1 derives to $DLuv$ and $wDuL$ but also to $DLuL$ and $wDuv$.

In each case the second pair of strings does not have a correspondence in SEQ^B and will cause the global condition to fail. The following informally stated transformation rules eliminate them: Consider a set of productions with symbols $X^1, X^2, \ldots, X^k$, $2 \leqq k \leqq m_x$ on their lefthand side. For an inherited (synthesized respectively) attribute of X determine the largest (smallest) index i such that $X^i ::= pLq\,(X^i ::= pDq)$. Eliminate $L(D)$ from all productions for X^j with $j \neq i$. Insert L in front (D at the end) of all productions for X^i which do not yet contain $L(D)$. By this means the L and D symbols are adjusted to the same section of the visit-sequences enlarging the lifetime of the corresponding attribute. (For ease of description we here assumed that only one $L(D)$ symbol contain in the productions derived from one visit-sequence. In fact one has to consider separately the symbols corresponding to different attribute occurrences.) The above examples are then transformed to

a)

$$\begin{array}{ll} Z^1 ::= DX^1\, u X^2 & \\ X^1 ::= v & X^2 ::= L \\ X^1 ::= \varepsilon & X^2 ::= Lw \end{array}$$

Z^1 derives to $DvuL$, $DuLw$, $DvuLw$, DuL

b)

$$\begin{array}{ll} Z^1 ::= X^1\, u X^2 & \\ X^1 ::= DY^1 & X^2 ::= Y^2 \\ X^1 ::= vD & X^2 ::= Y^1\, w\, Y^2 \\ Y^1 ::= s & Y^2 ::= L \end{array}$$

Z^1 derives to $DsuL$, $vDuswL$, $DsuswL$, $vDuL$

c)

$$\begin{array}{ll} Z^1 ::= X^1\, u X^2 & \\ X^1 ::= Y^1 & X^2 ::= Lv \\ X^1 ::= w\, Y^1 & X^2 ::= L \\ Y^1 ::= D & \end{array}$$

Z^1 derives to $DuLv$, $wDuL$, DuL, $wDuLv$

Again neither of the last two strings corresponds to SEQ^B. But now D and L symbols are paired, and if the symbols are not "shifted over strings" which produce non-empty strings, then the result will be less pessimistic.

Finally we show that the class of languages SEQ defined by a visit-oriented evaluator (e.g. our lifetime interpreter) is greater than the class of context-free languages. In other words: It is an intrinsically context-sensitive property of the evaluator operating on the tree structure. Hence Lemma 1 cannot be strengthened by a more sophisticated construction of G_L in order to achieve $L(G_L)=SEQ$ in general.

This property is easily demonstrated by an evaluator producing $SEQ=a^n b^n c^n$. Let the original grammar G be

$$S::=X \qquad X::=tX \qquad X::=\varepsilon$$

Assume that there are visit-sequences (omitted here) which are turned by our transformation into G_L:

$$\begin{array}{lll} S^1::=X^1\,X^2 & & (vsl_{p1}) \\ X^1::=a\,X^1\,b & X^2::=c\,X^2 & (vsl_{p2}) \\ X^2::=\varepsilon & X^2::=\varepsilon & (vsl_{p3}) \end{array}$$

with some terminal strings for a, b, and c. Obviously $L(G_L)=a^l b^m c^n$, but $SEQ=a^n b^n c^n$. Of course any context-free language can be produced by a trivial one-visit evaluator.

4. Conditions for Global Stacks

If the conditions for mapping an attribute to a global variable fail one can check whether the lifetimes of its instances are not overlapping, i.e. they are either disjoint or properly included. In that case the attribute instances can be implemented by a global stack. The storage for the sum of all instances of that attribute in a tree is then reduced to the amount needed for the maximum number of instances existing at the same time – the depth of the stack. We present a sufficient condition based on the lifetime visit-sequences of Sect. 3. It is shown that all attributes for which the stack condition holds can be implemented by one single stack or any groups of attributes mapped to several stacks.

In contrast to the global variable condition of Sect. 3 we can not base the stack condition on the language of the grammar G_L: The marks for begin and end of lifetime (D, L) are not distinguished for different instances. Hence for two instances i, j one can not decide whether $uDDLLv \in L(G_L)$ has to be interpreted as $uD_i\,D_j\,L_j\,L_i v$ (properly included lifetimes) or as $uD_i\,D_j\,L_i\,L_j v$ (overlapping lifetimes). Figure 7 demonstrates this situation by an example for a visit trace which reaches two tree nodes i and j twice. In other words: By context-free means one can not distinguish whether attribute evaluation performs a tree walk from a tree node i labelled X to a different node j with the same label or back to the same node j.

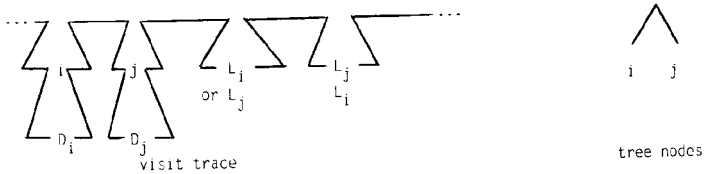


Fig. 7. Example for DDLL caused by nested or overlapped lifetimes

Hence a more pessimistic condition on the structure of the lifetime visit-sequences is stated. The basic idea is best explained by comparing it with the well-known principle of runtime stacks for recursive procedure calls:
A visit-sequence vs_p associated to a production P: $X_0 ::= X_1 \ldots X_n$ is separated into sections each ending with an ancestor visit. Such a section is considered as a procedure called by corresponding descendant visits. Within the procedure storage is allocated for local entities – the stack attributes of visited symbol occurrences $X_1, \ldots, X_n$ – and released after being used. Hence an attribute can be implemented by a global stack if the lifetime of its instances does not exceed such a section. This property is easily checked using the lifetime visit-sequences.

The placement of the allocation and deallocation operation within one section enlarges the duration of space allocation for the attribute compared with its lifetime. This will cause the stack to be deeper than necessary in some cases. (This technique is called "stacking from above" in [6].) On the other hand it relaxes the stack condition (some overlaps are turned into proper inclusions) and allows to combine arbitrary stackable attributes on one stack. Furthermore the pairs of corresponding push- and pop-operations are contained in the same context of one visit-sequence. Up to now in the GAG-System these operations are associated with the exact lifetime. (The consequences for GAG are more but smaller stacks and violations of the stack condition in some more cases as discussed e.g. in [6].)

For pass-oriented AGs this method is well-known: Attributes used only in a single pass can be implemented by a stack (cf. [5]). Here it is generalized to visit-oriented evaluators for larger AG-classes: Attributes used only during one visit of a subtree are stackable.

We first give an outline of the problems to be solved subsequently:

a) Decide whether a given attribute $X.a$ can be implemented by a stack.
b) Insert operations for allocation (A) and release (R) of stackspace for $X.a$ into visit-sequences containing visits to an occurrence of X. The lifetime of $X.a$ in that context has to be bracketed by A and R.
c) Assure stack discipline and proper access to stack elements in cases where more than one attribute occurrence is stacked in the same context (i.e. X occurs more than once on the right hand side of a production).
d) Map different attributes to the same stack.

Let us first demonstrate the problem by a modification of our example grammar before we present a general solution: If we replace the attribute rule (a24)

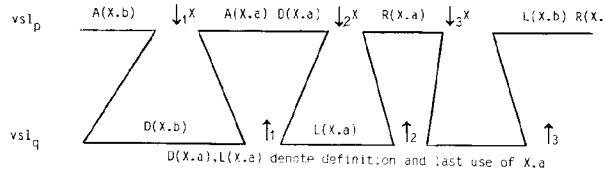


Fig. 8. Insertion of stack operations A and R

in Fig. 1 by

(a 24) $$Y_1.d := f_5(Y_1.c,\ Y_2.d)$$

then lifetime of $Y.c$ is lengthened. vs_{p2} and vsl_{p2} are modified correspondingly. In the lifetime grammar G_L of Fig. 6 the first production for Y^2 is replaced by

$$Y^2 ::= D_a\ Y^1\ L_b\ D_c\ Y^2\ L_c\ L_d\ D_d.$$

Now the language of G_L is

$$L(G_L) = (D_a\ L_a\ D_b\ L_b\ D_c)^n\ L_c\ D_d (L_c\ L_d\ D_d)^{n-1}\ L_d.$$

Hence $L(G_L^B) = D^n\ L^n$ for $B = c$, and the global variable condition does not hold for c (globalization of a, b, and d is not affected).

Now consider the lifetime visit-sequences with respect to $Y.c$ only (the potentially stackable attribute). One verifies that the lifetime of $Y.c$ does not exceed a single section of vsl_{p1} or vsl_{p2} respectively. Hence we can insert allocation operations A and R as follows (attributes a, b, and d are omitted):

$$vsl_{p1}: \downarrow_1 Y,\ A(Y.c),\ \rightarrow Y.c,\ \downarrow_2 Y,\ R(Y.c),\ \uparrow_1$$
$$vsl_{p2}: \uparrow_1,\ \downarrow_1 Y_2,\ A(Y_2.c),\ \rightarrow Y_2.c,\ \downarrow_2 Y_2,\ R(Y_2.c),\ Y_1.c,\ \uparrow_2$$
$$vsl_{p3}: \uparrow_1,\ Y.c,\ \uparrow_2$$

Obviously the allocation operations include the lifetimes of $Y.c$ instances, and stack discipline is obeyed.

We now consider the general case. For ease of presentation we first consider a single attribute $X.a$ for globalization. Furthermore we assume that there is no production with more than one occurrence of X on the right hand side. The simplest way to solve problem (a) is to insert $A(X.a)$ and $R(X.a)$ into the visit-sequences, and then check whether the pairs lie each in one visit-sequence section.

Lifetimes of instances of $X.a$ are determined by references to $X.a$ within pairs of visit-sequences associated to productions p: $Y ::= uXv$ and q: $X ::= w$, as shown in Figs. 2 and 3.

Figure 8 contains one complete section of vsl_p interacting with three sections of vsl_q. Definition and use of an inherited attribute $X.a$ and a synthesized $X.b$ are shown. If $X.a$ is stackable and its lifetime is included in that section of vsl_p the allocation operations will be inserted there. vsl_p contains defining

(applied) occurrences of $X.a$ ($X.b$) and vice versa for vsl_q (if the AG is in Bochmann Normal Form; otherwise both may contain applied occurrences). In order to map the lifetime to vsl_p we say a visit $\downarrow_j X$ references $X.a$ if there is a q such that its j-th section contains a reference to $X.a$. By that means we ensure that our lifetime considerations for vsl_p hold for any possible interacting vsl_q.

For an inherited (synthesized) attribute $X.a$ and all productions of the form q determine the smallest number k (and the largest number l) such that the k-th (the l-th) section of a vsl_q contains a reference to $X.a$. Then the visits $\downarrow_k X$ upto $\downarrow_l X$ must be included by the $A-R$ pair in vsl_p. Now in each vsl_p insert $A(X.a)$ as late as possible but not after the first reference to $X.a$ and $\downarrow_k X$; and insert $R(X.a)$ as early as possible but not before the last reference to $X.a$ and $\downarrow_j X$.

The **global stack condition** is stated on the modified visit-sequences: *$X.a$ is implemented by a global stack if each pair of corresponding $A(X.a)$ and $R(X.a)$ operations is contained in one visit-sequence section.*

The condition ensures that during the allocation time of an instance of $X.a$ the subtree of the associated node is not left by an ancestor visit. If the condition does not hold the A- and R-operations are removed and $X.a$ is discarded for globalization.

Obviously the stack condition holds for all attributes if there is exactly one visit to each symbol. This result corresponds to stack allocation in pass-oriented evaluators for attributes used only in one pass.

For the solution of problem (c) we assume that $X.a$ is stackable and there is a production p where X occurs more than once on the right hand side, e.g. p: $Y::=u_1 X u_2 X v$. In order to ensure the stack discipline we distinguish three cases:

1) Corresponding A- and R-operations for both occurrences of X are contained in different sections of vsl_p, i.e. each set of visits $\downarrow_k X$ to $\downarrow_l X$ is contained in different sections. In this case stack discipline is achieved without modification.
2) No modification is needed too, if corresponding A- and R-operations are contained only in one section and if they are properly bracketed.
3) If some A and R pairs in one section are not properly bracketed we can lengthen the allocation time for certain attributes: In order to avoid overlapping allocations: A-operations can be shifted to the left and/or R-operations can be shifted to the right until stack discipline is achieved.

In the above cases (2) and (3) there are areas within the visit-sequence section where more than one stack element can be referenced.

Let vsl_p be for example the sequence

$$v_1\, A(X_1.a)\, v_2 \downarrow_1 X_1\, v_3\, A(X_2.a)\, v_4 \downarrow_1 X_2\, v_5 \downarrow_2 X_1\, v_6 \downarrow_2 X_2\, v_7\, R(X_2.a)\, R(X_1.a)\, v_8.$$

Then references to $X_1.a$ within v_2 or v_3 access top of stack. If they occur in $v_4, \ldots, v_7$ top of stack minus one has to be accessed. For $X_2.a$ always top of stack has to be accessed. In any case for references within vsl_p the relative stack position can be determined statically.

More care has to be taken for references reached by the visits to the occurrences of X: Within visit-sequences q reached by those visits the particular

stack context of p can not be distinguished (e.g. $\downarrow_2 X_1$ and $\downarrow_2 X_2$ in the above example). For these references it is assumed that $X.a$ is on top of the stack. Hence we have to adjust the stack within vsl_p before and after visits referencing an $X.a$ which is not on top of stack ($\downarrow_2 X_2$ in the example). We can do that by simply pushing a copy of the referenced attribute before the visit and popping it afterwards. (If the visit references a defining occurrence the original stack element has to be assigned from the copy after the visit.) This technique allows to stack attributes whose lifetime does not span an ancestor visit according to the global stack condition.

Now the combination of stacks for different attributes (problem (d)) can be achieved rather easily: Consider two arbitrary stackable attributes $X.a$ and $Y.b$. In order to combine the two stacks we can ensure stack discipline and proper stack access by the same means as we presented for different occurrences of one attribute in (c).

A special situation arises if we combine the stacks for two attributes $X.a$ and $X.b$ of one symbol X: The assumption for visit-sequences q to access the attribute from top of the stack can not be true for $X.a$ and $X.b$ if both are referenced in one visit-sequence section. In that case we can conclude that all visits referencing $X.a$ or $X.b$ always occur in one section. Otherwise either $X.a$ or $X.b$ could not be stackable. Hence we can fix a certain order on the stack for instances of $X.a$ and $X.b$ (e.g. $X.a$ on top and $X.b$ on top minus one). This order is then established in vsl_p either by shifting A- and R-operations or additional copies as discussed above.

Finally it should be mentioned that for storage savings achieved by attribute stacks a price has to be paid: Runtime and code length is increased by the introduced stack operations. (That is not the case for global variables.) However, in certain situations identical attributes assignments between topmost stack elements and the inserted stack operations can be eliminated.

5. Conclusion

In this paper we presented a method for storage improvements for attribute evaluators. Since the lifetime analysis is based only on statically determinable properties of the AG the method is well suited for generating systems, i.e. attribute evaluator generators.

In the GAG-System [8] an attribute optimization phase is implemented which is based on the underlying idea of the method presented here. There the visit-sequences are analyzed directly without a transformation into context-free grammars. (The latter idea is roughly presented in a system overview in [10].) Whereas the conditions used in that implementation are even stronger than those presented here it yields significant storage reductions (see [8].) Apart from the additional improvements expected in practical cases this method based on language analysis is more comprehensible and provable. The method will be used in the implementation of a successor system for GAG. Besides the more systematic technique it will allow to quantify the improvement for realistic AGs.

We did not discuss the aspect of modification of visit-sequences in order to improve attribute globalization. It would be beyond the scope of this paper and is a topic of further research. A different application of a simplified version of our lifetime analysis method is presented in [11]: For a set of mutually recursive procedures it is decided which parameters can be allocated statically.

References

1. Engelfriet, J., File, G.: Simple multi-visit attribute grammars. J. Comput. Syst. Sci. **24**, 283–314 (1982)
2. Kastens, U.: Ordered Attribute Grammars. Acta Inf. **13**, 229–256 (1980)
3. Ganzinger, H.: On Storage Optimization for Automatically Generated Compilers. Lect. Notes Comput. Sci. Vol. 67, pp 132–141. Berlin, Heidelberg, New York: Springer 1979
4. Räihä, K.-J.: Dynamic allocation of space for attribute instances in multi-pass evaluators of attribute grammars, in Proc. SIGPLAN Symp. Compiler Construction, SIGPLAN Notices **14**, 26–38 (1979)
5. Jazayeri, M., Pozefsky, D.: Space-Efficient Storage Management in an Attribute Grammar Evaluator. ACM TOPLAS **3**, 388–404 (1981)
6. Farrow, R., Yellin, D.: A Comparison of Storage Optimizations in Automatically-Generated Attribute Evaluators. Acta Inf. **23**, 393–427 (1986)
7. Sonnenschein, M.: Global Storage Cells for Attributes in an Attribute Grammar. Acta Inf. **22**, 397–420 (1985)
8. Kastens, U., Hutt, B., Zimmermann, E.: GAG: A Practical Compiler Generator. Lect. Notes Computer Sci. Vol. 141. Berlin, Heidelberg, New York: Springer 1982
9. Waite, W.M., Goos, G.: Compiler Construction. Berlin, Heidelberg, New York: Springer 1983
10. Kastens, U.: The GAG-System – A Tool for Compilar Construction. In Lorho, B. (ed) Methods and Tools for Compiler Construction, pp. 165–181. Cambridge: University Press
11. Kastens, U., Schmidt, M.: Lifetime Analysis for Procedure Parameters, in European Symposium of Programming. Lect. Notes Computer Sci. Vol. 213. Berlin, Heidelberg, New York: Springer 1986

Appendix

The following attribute grammar defines a small block structured and expression oriented example language. In order to concentrate upon results of lifetime analysis syntax and static semantics are reduced to a skeleton definition only. The attribute rules shall give a rough idea of scope rules and typing specification. They are not completely elaborated and explained here because only their structure is important.

The following attributes are used:

symbols	**attributes**		
Prog			
Block		*env*	*tp*
Decls	*dec*	*env*	
Decl	*dec*	*env*	
Body		*env*	*tp*
Expr		*env*	*tp*
Term		*env*	*tp*

env describes the set of definitions valid in the environment of the symbol
dec describes the set of definitions introduced in the subtree of the symbol
tp describes the type of the construct represented by the symbol

(p1)	$Prog ::= block$	(a11)	$Block.env \leftarrow \varnothing$
(p2)	$Block ::= (Decls\ Body)$	(a21)	$Decls.env \leftarrow Decls.dec \cup Block.env$
		(a22)	$Body.env \leftarrow Decls.dec \cup Block.env$
		(a23)	$Block.tp \leftarrow Body.tp$
(p3)	$Decls_1 ::= Decls_2\ Decl;$	(a31)	$Decls_1.dec \leftarrow Decls_2.dec \cup Decl.dec$
		(a32)	$Decls_2.env \leftarrow Decls_1.env$
		(a33)	$Decl.env \leftarrow Decls_1.env$
(p4)	$Decls ::=$	(a41)	$Decls.dec \leftarrow \varnothing$
(p5)	$Decl ::= typeid\ id = Expr$	(a51)	$Decl.dec \leftarrow def(id, typeid)$
		(a52)	$Expr.env \leftarrow Decl.env$
(p6)	$Body_1 ::= Expr\ ;\ Body_2$	(a61)	$Expr.env \leftarrow Body_1.env$
		(a62)	$Body_2.env \leftarrow Body_1.env$
		(a63)	$Body_1.tp \leftarrow Body_2.tp$
(p7)	$Body ::= Expr$	(a71)	$Expr.env \leftarrow Body.env$
		(a72)	$Body.tp \leftarrow Expr.tp$
(p8)	$Expr_1 ::= Expr_2 + Term$	(a81)	$Expr_2.env \leftarrow Expr_1.env$
		(a82)	$Term.env \leftarrow Expr_1.env$
		(a83)	$Expr_1.tp \leftarrow oprid(Expr_2.tp, Term.tp)$
(p9)	$Expr ::= Term$	(a91)	$Term.env \leftarrow Expr.env$
		(a92)	$Expr.tp \leftarrow Term.tp$
(p10)	$Term ::= id$	(a101)	$Term.tp \leftarrow ident(id, Term.env)$
(p11)	$Term ::= Block$	(a111)	$Block.env \leftarrow Term.env$
		(a112)	$Term.tp \leftarrow Block.tp$

In the following visit-sequences the *env* and *tp* attributes are computed in one visit. For *Decls* and *Decl* a preceeding visit is needed collecting the local definitions. (For better readability all indices 1 are omitted if no corresponding 2 exists.)

(vs_1) $\varnothing \rightarrow Block.env, \downarrow Block, \uparrow$

(vs_2) $\downarrow_1 Decls, Decls.dec \cup Block.env \rightarrow Decls.env, \downarrow_2 Decls,$
$Decls.dec \cup Block.env \rightarrow Body.env, \downarrow Body, Body.tp \rightarrow Block.tp, \uparrow$

(vs_3) $\downarrow_1 Decls_2, \downarrow_1 Decl, Decls_2, dec \cup Decl.dec \rightarrow Decls_1.dec, \uparrow_1,$
$Decls_1.env \rightarrow Decls_2.env, \downarrow_2 Decls_2, Decls_1.env \rightarrow Decl.env, \downarrow_2 Decl, \uparrow_2$

(vs_4) $\varnothing \rightarrow Decls.dec, \uparrow_1, \uparrow_2$

(vs_5) $def(id, typeid) \rightarrow Decl.dec, \uparrow_1, Decl.env \rightarrow Expr.env, \downarrow Expr, \uparrow_2$

(vs_6) $Body_1.env \rightarrow Expr.env, \downarrow Expr, Body_1.env \rightarrow Body_2.env, \downarrow Body_2,$
$Body_2.tp \rightarrow Body_1.tp, \uparrow$

(vs_7) $Body.env \rightarrow Expr.env, \downarrow Expr, Expr.tp \rightarrow Body.tp, \uparrow$

(vs_8) $Expr_1.env \rightarrow Expr_2.env, \downarrow Expr_2, Expr_1.env \rightarrow Term.env, \downarrow Term,$
$oprid\ (Expr_2.tp, Term.tp) \rightarrow Expr_1.tp, \uparrow$

(vs_9) $Expr \rightarrow Term.env, \downarrow Term, Term.tp \rightarrow Expr.tp, \uparrow$

(vs_{10}) $ident\ (id, Term.env) \rightarrow Term.tp, \uparrow$

Lifetime analysis according to the previous sections yields the following results: The global variable condition holds for

$$Term.env, Decl.env, Decl.dec, Decls.dec, Term.tp, Body.tp, Block.tp$$

For all other attributes the global stack condition holds.

The sets $\{Term.tp, Body.tp, Block.tp\}$ and $\{Term.env, Decl.env\}$ can be mapped to a single variable each. Then the attribute rules (a23), (a63), (a112) can be omitted (identical assigns).