Using cached functions and constructors for incremental attribute evaluation

Maarten Pennings, Doaitse Swierstra and Harald Vogt

RUU-CS-92-11
March 1992

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Using cached functions and constructors for incremental attribute evaluation

Maarten Pennings, Doaitse Swierstra and Harald Vogt

Technical Report RUU-CS-92-11
March 1992

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924-3275

Using cached functions and constructors for incremental attribute evaluation

Maarten Pennings, Doaitse Swierstra and Harald Vogt*

Abstract

This paper presents a technique for the efficient incremental evaluation of Attribute Grammars. Through its generality, the applied approach may be affective too in the evaluation of Higher-order Attribute Grammars.

Our approach is an extension of a simpler algorithm for incremental evaluation, where functions, corresponding to visit sequences, are cached. Consequently, attributes are now either found in the cache or they are recomputed, so there is no longer need to represent the attributed tree explicitly. We may share common subtrees, avoiding repeated attribute evaluation, thus solving a typical HAG problem.

We propose the following change: instead of explicitly representing the tree and calling visit sequence functions to compute the attributes, the tree is represented *through* a set of visit functions corresponding to the successive visits. These functions are constructed using the visit sequences as building blocks.

This technique has two major advantages. Firstly, a visit function characterizes precisely that part of the tree that would actually have been visited in the previous approach, thus increasing the number of cache hits. Secondly, copy rules may be removed during the construction phase. This results in shortcircuiting copychains and in minimizing the number of recomputed functions.

1 Introduction

Attribute Grammars (AGs) [Kn68, Kn71] describe the computation of attributes: values associated with the nodes of a tree. Trees are described with a context free grammar and the attribute computation is defined through semantic functions. Attribute grammars are used to define languages and form the basis of compilers, language-based editors and other language based tools [DeJoLo88, DeJo90, Al91].

Higher order attribute grammars (HAGs) [VoSwKu89] remove the artificial distinction between the syntactic level (context free grammar) and the semantic level (attributes) in attribute grammars. This strict separation is removed in two ways: trees can be defined through

*Department of Computer Science, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands, E-Mail: maarten@cs.ruu.nl, doaitse@cs.ruu.nl and harald@cs.ruu.nl.

attribution, and such trees may be instantiated (and attributed!). Trees defined through attribution are known as non-terminal attributes. A non-terminal attribute occurs both as a non-terminal on the right hand side of a production rule and as an attribute on the left hand side of an attribution rule of that production, thus promoting trees to first class citizens.

Our new evaluation technique is based on an efficient algorithm for ordered HAGs which is presented in the next section. In Section 3 the new technique is explained with an informal example. Section 4 presents a detailed example using the old algorithm and in Section 5 the new technique is applied to it. Finally, Section 6 contains the conclusions.

2 Introduction to the old algorithm

The new algorithm presented in this paper resulted from our research on efficient incremental evaluators for higher order attribute grammars. Typical HAG features make the approaches for normal AGs unsuitable for evaluating them.

Attribute evaluators for ordered AGs [Ka80, Ye83, TeCh90] can be trivially adapted to handle ordered HAGs [VoSwKu89]. The adapted evaluator, however, attributes instances of non-terminal attributes with the same inherited attributes separately. This leads, amongst others, to a non-optimal incremental behavior after a change to such an attribute, as can be seen in the recently published algorithm of [TeCh90]. A better technique is the evaluation algorithm for OHAGs in [VoSwKu91, SwVo91]. It handles multiple occurrences of the same subtree efficiently. This algorithm is based on the combination of the following ideas:

Visit sequence functions. Attribute values are computed by visit sequence functions. Such functions take as parameter a tree and a subset of inherited attributes of the root of that tree and they return a subset of the synthesized attributes (of the root). The entire evaluator consists of visit sequence functions that recursively call each other in order to attribute the tree.

Caching. In a conventional incremental treewalk evaluator a partially attributed tree can be considered as a very efficient caching mechanism—where cashing is replaced by explicit navigation—for the semantic functions. Instead of using a separate cache for the results of semantic functions, as was done in [Pu88] only visit sequence functions are cached: one uniform treatment of semantic functions and attribute evaluation. Furthermore, we have no seperate administration on whether attributes have changed and further visits are necessary.

This approach is more efficient because a cache hit of a visit sequence function means that the entire visit to the (possibly large) tree can be skipped. Furthermore, a visit sequence function may return the results of several semantic functions at a time.

Memoed constructors. Since attributes may be found in the cache, there is no longer need to store them in the tree. This allows us to share multiple instances of the same tree. As in [TeCh90], we use memoed treeconstructors. A memoed (cached) constructor is called a 'hashing cons' in [Hu85].

Bindings. Although the above ideas seem appealing at first sight, a complication is that attributes computed in an earlier visit may have to be preserved for use in later visits.

Normally, this is no problem since attributes are stored in the tree. Now these values, called *bindings*, must be passed explicitly to the future visits. Each visit sequence function therefore not only computes synthesized attributes but also bindings for subsequent visits. Bindings computed by earlier visits are passed as parameters to later visit sequence functions.

3 Introduction to the new algorithm

The original visit sequences of [Ka80] were designed with the goal to minimize the number of visits to each node. In the case of incremental evaluation, however, one's goal will be to maximize the number of cache hits for the visit sequence functions. The parameters of these functions—the tree, the inherited attributes and the bindings—form the cache key. Let us examine them.

An essential property of the construction of bindings is that when calling a visit sequence function with its bindings, these bindings contain *precisely* that information that will be used in this visit. This is a direct result of the fact that these bindings were constructed during earlier visits, at which time it was known what productions had been applied and what actual dependencies are occurring in the subtrees. Thus there is little room for improvement here.

The situation is different however when we inspect the role of the tree parameter to the visit sequence functions. Always the complete tree is passed and not only those parts that will actually be traversed by the called visit sequence function. Since the complete tree is used as part of the key in the function cache, unnecessary visit calls may be performed. Our new technique eliminates the above mentioned shortcoming by modifying the [VoSwKu91] algorithm as follows: instead of first constructing the tree followed by applying the visit sequence function associated with the root to the entire tree, we *represent* the tree through a set of large visit functions. These functions are constructed by composing the visit sequence functions of each treenode. As a result, the number of cache hits increases since the visit functions now only depend on that part of the tree that will actually be visited.

Let us have a look at an example where we have visits which pass through different parts of the subtree. We model a language which does not demand identifiers to be declared before they are used. This naturally leads to a two-pass algorithm: one pass for constructing the environment and a second pass for actually compiling the statements. In Figure 1 an example tree is given. The dashed arrows indicate the dataflow: the leftmost for collecting the declarations, the middle one for distributing the environment and the rightmost for computing the code. Notice that for collecting the declarations only the identifiers of the declaration nodes are passed whereas for computing the code only the identifiers of the statement nodes are visited.

What happens when we change a using occurrence? Suppose we change the N-node labeled with **Change**. Due to constructor memoing, the newly constructed tree shares the lower part (the four L-nodes and three N-nodes) with the old tree. But there is more: since the first visit subsequence of a **stat** production does not refer to the N-son (a **stat** production doesn't add a declaration), the first visit function isn't changed. So the *entire* first visit of the new tree (called from the **root**-function) is found in the cache and hence may be skipped.

Not only doesn't the first visit function to a **stat** node refer to its N-son, it doesn't do much

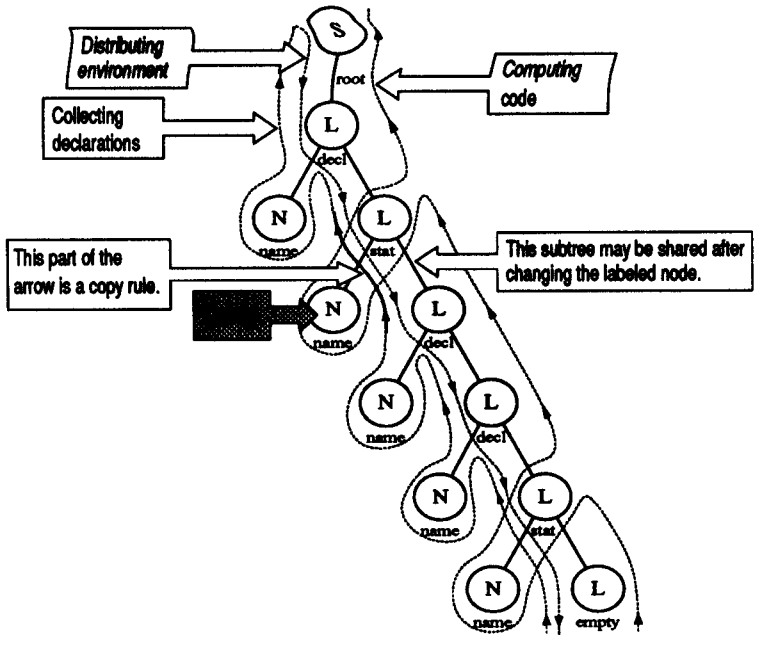

Figure 1: Dataflow analyses.

at all. It simply copies values, or rephrased, it consists of so called *copy rules* only. So, If we add another statement just above the one we just changed, a useless identity function is inserted in the first visit function. Hence this function is changed, so no cache hit will occur. But, if we are able to identify copy rules (identity functions) and *eliminate* them, there will be a cache hit again. In this way, we not only get a smaller (and faster) visit function, but we will also get more cache hits.

The next section explains the old algorithm by using the above example in detail. This example will also be used in the explanation of the new algorithm in Section 5.

4 The old algorithm: an example

We present a simple grammar implementing the "programming language" from the previous section. To illustrate all aspects of the algorithm we made the grammar two-pass. In pass one the definitions will be collected, so that in pass two the actual translation may take place. Thus the "main" non-terminal (L) has two visits, so that the visit sequences [Ka80] associated with the production rules that may be applied to L (**stat**, **decl** and **empty**) consist of two subsequences.

We now define visit sequence functions corresponding to these visit subsequences. For each non-terminal we construct one function for each visit. The ith visit sequence function of a non-terminal is applied to the inherited attributes which have newly become available for visit i and it returns the synthesized attributes for that visit.

This doesn't differ too much from Kastens' approach. But Kastens uses the abstract tree as

a repository in which attributes are stored between defining and using visits. As we do not have an explicit tree representation we must store these intermediate results somewhere else. The problem is solved by introducting so called ***bindings***, their usage is illustrated further on.

4.1 The example grammar

A program in the example grammar is a list of "declarations" (such as **var x**) and "statements" (like **use x**). They may be mixed freely, and we do not require *definition before use*. In the translation process, each variable is mapped onto a number. The resulting list will contain this number for each using occurrence, and the negation of it for the defining occurrence. Hence

```
(use x; use y; var y; use x; use y; use z; var z; use x; var x; use x;)
```

is mapped onto the list

```
3, 1, -1, 3, 1, 2, -2, 3, -3, 3
```

The grammar of our language has the following (labeled) production rules:

root : $S \rightarrow (L)$
decl : $L \rightarrow$ **var** $N; L$
stat : $L \rightarrow$ **use** $N; L$
empty : $L \rightarrow \epsilon$
name : $N \rightarrow str$

Since we are using trees as arguments for our visit sequence functions, we need a concise notation for trees. We will follow a MIRANDA-like notation [Tu85] for terms. So, the above grammar is transformed (leaving out the terminals ('(', ')', '**var**', '**use**', and ';') since they are not of interest to us) to:

$S =$ **root**(L)
$L =$ **decl**(N, L)
  | **stat**(N, L)
  | **empty**$()$
$N =$ **name**(str)

This grammar is as follows augmented with attributes. Apart from the already introduced type *str* (representing identifiers) we distinguish $env = [str]$ and $code = [num]$ where type *num* represents the natural numbers (with square brackets we denote "list-of"). The start-symbol S has a single (synthesized) attribute returning the code whereas the listsymbol L has three attributes: one that collects the declarations (synthesized), one that distributes the environment (inherited) and one that synthesizes the code. See Figure 2 for the complete attribute grammar.

4.2 Visit (sub)sequences

In Figure 3 the attribute dependencies are presented graphically. The attributes of the symbols are topologically sorted according to their (indirect) dependencies. The dashed lines

Signature
$S\langle\uparrow code\rangle$
$L\langle\uparrow env, \downarrow env, \uparrow code\rangle$
$N\langle\uparrow str\rangle$

$S\langle\uparrow c_0\rangle = \mathbf{root}(L\langle\uparrow d, \downarrow e, \uparrow c_1\rangle)$
$\quad e := d;$
$\quad c_0 := c_1$
$N\langle\uparrow n\rangle = \mathbf{name}(str)$
$\quad n := str$

$L\langle\uparrow d_0, \downarrow e_0, \uparrow c_0\rangle = \mathbf{decl}(N\langle\uparrow n\rangle, L\langle\uparrow d_1, \downarrow e_1, \uparrow c_1\rangle)$
$\quad d_0 := [n] +\!\!+ d_1;$
$\quad e_1 := e_0;$
$\quad c_0 := [-\text{lookup} \cdot e_0 \cdot n] +\!\!+ c_1$
$\mid \mathbf{stat}(N\langle\uparrow n\rangle, L\langle\uparrow d_1, \downarrow e_1, \uparrow c_1\rangle)$
$\quad d_0 := d_1;$
$\quad e_1 := e_0;$
$\quad c_0 := [\text{lookup} \cdot e_0 \cdot n] +\!\!+ c_1$
$\mid \mathbf{empty}()$
$\quad d_0 := [\,];$
$\quad c_0 := [\,]$

Figure 2: The attribute grammar.

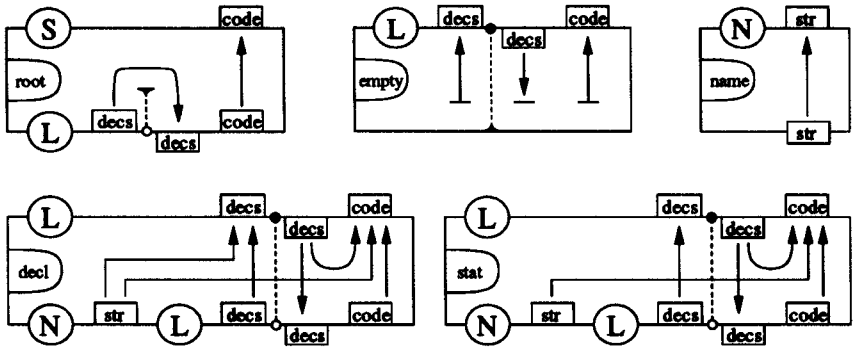

Figure 3: Attribute dependencies in a production rule.

indicate a visit border: attributes to the left of it are evaluated during the first visit and attributes to the right of it during the second.

Note that when a tree is constructed from these productions, the "pieces" fit nicely together. This does not only hold for the non-terminals, the inherited and synthesized attributes, but also for the visit borders: a solid disk hooks into an open circle splitting the entire tree. With this division it is fairly simple to determine suitable visit (sub)sequences: see Figure 4 in which **name** and **empty** are omitted for brevity.

4.3 From visit (sub)sequences to a functional program

We will not show the mapping from visit (sub)sequences to visit sequence functions. This is a straightforward process: each visit(X, i) instruction is mapped to a visit-X-i function call with the tree (X) as first parameter and the appropriate inherited attributes as the next parameters. The function returns the computed synthesized attributes. In Figure 5 the resulting functions are given.

The reader may have noticed that we use $\cdot$ for function application. We do this in order not

$$
\begin{array}{l}
\mathrm{VS}(\mathbf{root}(L)) \\
\quad = \mathrm{VSS}(\mathbf{root}(L), 1) \\
\qquad = \mathrm{visit}(L, 1)\ \{\mathrm{ret}\ d\} \\
\qquad\ ; e := d \\
\qquad\ ; \mathrm{visit}(L, 2)\ \{\mathrm{ret}\ c_1\} \\
\qquad\ ; c_0 := c_1 \\
\qquad\ ; \mathrm{return}(1)
\end{array}
\qquad
\begin{array}{l}
\mathrm{VS}(\mathbf{decl}(N, L)) \\
\quad = \mathrm{VSS}(\mathbf{decl}(N, L), 1) \\
\qquad = \mathrm{visit}(N, 1)\ \{\mathrm{ret}\ n\} \\
\qquad\ ; \mathrm{visit}(L, 1)\ \{\mathrm{ret}\ d_1\} \\
\qquad\ ; d_0 := [n] +\!\!+ d_1 \\
\qquad\ ; \mathrm{return}(1) \\
\quad ; \mathrm{VSS}(\mathbf{decl}(N, L), 2) \\
\qquad = e_1 := e_0 \\
\qquad\ ; \mathrm{visit}(L, 2)\ \{\mathrm{ret}\ c_1\} \\
\qquad\ ; c_0 := [-\mathrm{lookup} \cdot e_0 \cdot n] +\!\!+ c_1 \\
\qquad\ ; \mathrm{return}(2)
\end{array}
\qquad
\begin{array}{l}
\mathrm{VS}(\mathbf{stat}(N, L)) \\
\quad = \mathrm{VSS}(\mathbf{stat}(N, L), 1) \\
\qquad = \mathrm{visit}(L, 1)\ \{\mathrm{ret}\ d_1\} \\
\qquad\ ; d_0 := d_1 \\
\qquad\ ; \mathrm{return}(1) \\
\quad ; \mathrm{VSS}(\mathbf{stat}(N, L), 2) \\
\qquad = \mathrm{visit}(N, 1)\ \{\mathrm{ret}\ n\} \\
\qquad\ ; e_1 := e_0 \\
\qquad\ ; \mathrm{visit}(L, 2)\ \{\mathrm{ret}\ c_1\} \\
\qquad\ ; c_0 := [\mathrm{lookup} \cdot e_0 \cdot n] +\!\!+ c_1 \\
\qquad\ ; \mathrm{return}(2)
\end{array}
$$

Figure 4: The visit sequences for productions **root**, **decl** and **stat**.

to confuse functions and constructors. Note also that we overload the symbols S, L and N; they are used as type identifiers as well as dummies in the patterns.

$$
\begin{array}{l}
\textbf{Signature} \\
\text{visit-S-1} \ :: S \qquad \rightarrow \mathit{code} \\
\text{visit-L-1} \ :: L \qquad \rightarrow \mathit{env} \\
\text{visit-L-2} \ :: L \times \mathit{env} \rightarrow \mathit{code} \\
\text{visit-N-1} \ :: N \qquad \rightarrow \mathit{str}
\end{array}
$$

$$
\begin{array}{l}
\text{visit-L-1} \cdot \mathbf{decl}(N, L) = d_0 \\
\quad \text{where } \boxed{\mathrm{n}} := \text{visit-N-1} \cdot N \\
\qquad\ ; d_1 := \text{visit-L-1} \cdot L \\
\qquad\ ; d_0 := [n] +\!\!+ d_1
\end{array}
$$

$$
\begin{array}{l}
\text{visit-L-1} \cdot \mathbf{stat}(N, L) = d_0 \\
\quad \text{where } d_1 := \text{visit-L-1} \cdot L \\
\qquad\ ; d_0 := d_1
\end{array}
$$

$$
\begin{array}{l}
\text{visit-S-1} \cdot \mathbf{root}(L) = c_0 \\
\quad \text{where } d := \text{visit-L-1} \cdot L \\
\qquad\ ; e := d \\
\qquad\ ; c_1 := \text{visit-L-2} \cdot L \cdot e \\
\qquad\ ; c_0 := c_1
\end{array}
$$

$$
\begin{array}{l}
\text{visit-L-2} \cdot \mathbf{decl}(N, L) \cdot e_0 = c_0 \\
\quad \text{where } e_1 := e_0 \\
\qquad\ ; c_1 := \text{visit-L-2} \cdot L \cdot e_1 \\
\qquad\ ; c_0 := [-\mathrm{lookup} \cdot e_0 \cdot \boxed{\mathrm{n}}] +\!\!+ c_1
\end{array}
$$

$$
\begin{array}{l}
\text{visit-L-2} \cdot \mathbf{stat}(N, L) \cdot e_0 = c_0 \\
\quad \text{where } n := \text{visit-N-1} \cdot N \\
\qquad\ ; e_1 := e_0 \\
\qquad\ ; c_1 := \text{visit-L-2} \cdot L \cdot e_1 \\
\qquad\ ; c_0 := [\mathrm{lookup} \cdot e_0 \cdot n] +\!\!+ c_1
\end{array}
$$

Figure 5: Visit sequence functions with side-effects: attribute $\boxed{\mathrm{n}}$ is stored in the tree.

4.4 Bindings

One of the major drawbacks of the visit sequence functions presented in Figure 5 is that they have side-effects. Some attributes must remain known over several visits. In our example $N.\mathit{str}$—in Figure 5 these occurrences are boxed—needs to be stored in the tree.

Taking another look at the attribute dependencies of production **decl** in Figure 3, we observe that one arrow—from $N.\mathit{str}$ to $L_0.\mathit{code}$—is crossing a visit border. Attribute $N.\mathit{str}$ is evaluated (and needed) in the first visit but it is also needed in the second one. Therefore it can not be deleted when returning from the first visit. The conventional solution is to store the attribute in the tree, but we take a different approach. The later needed values are passed to the father, and we rely on him to pass them down for the next visit. Such a link from visit-X-i to visit-X-j via a parent is called a binding from i to j for symbol X.

In Figure 6 the production rules are enhanced with bindings, *bind* = **stack of** *str*. Note that there is no need to bind $N.\mathit{str}$ in production **stat**: $N.\mathit{str}$ is not used in the first visit to L so N is visited in Ls second visit. That an arrow is still crossing the visit border in production

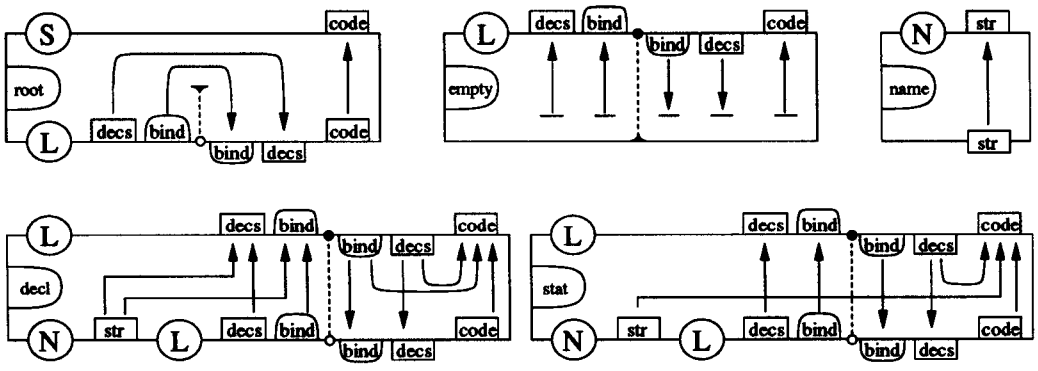

Figure 6: Introduction of bindings.

stat is just a consequence of the limited possibilities of 2D drawings. The visit sequence functions enhanced with bindings for our sample grammar are given in Figure 7.

Signature

visit-S-1 :: $S \rightarrow code$
visit-L-1 :: $L \rightarrow env \times bind$
visit-L-2 :: $L \times env \times bind \rightarrow code$
visit-N-1 :: $N \rightarrow str$

visit-S-1·**root**$(L) = c_0$
  where $\lceil d, b \rceil :=$ visit-L-1·L
  ; $e := d$
  ; $c_1 :=$ visit-L-2·L·e·b
  ; $c_0 := c_1$

visit-L-1·**decl**$(N, L) = \lceil d_0, b_0 \rceil$
  where $n :=$ visit-N-1·N
  ; $\lceil d_1, b_1 \rceil :=$ visit-L-1·L
  ; $d_0 := [n] +\!\!+ d_1$
  ; $b_0 :=$ push ·n·b_1

visit-L-2·**decl**(N, L)·e_0·$b_0 = c_0$
  where $e_1 := e_0$
  ; $\lceil n, b_1 \rceil :=$ pop ·b_0
  ; $c_1 :=$ visit-L-2·L·e_1·b_1
  ; $c_0 := [-\text{lookup} \cdot e_0 \cdot n] +\!\!+ c_1$

visit-L-1·**stat**$(N, L) = \lceil d_0, b_0 \rceil$
  where $\lceil d_1, b_1 \rceil :=$ visit-L-1·L
  ; $d_0 := d_1$
  ; $b_0 := b_1$

visit-L-2·**stat**(N, L)·e_0·$b_0 = c_0$
  where $n :=$ visit-N-1·N
  ; $e_1 := e_0$
  ; $b_1 := b_0$
  ; $c_1 :=$ visit-L-2·L·e_1·b_1
  ; $c_0 := [\text{lookup} \cdot e_0 \cdot n] +\!\!+ c_1$

Figure 7: Visit sequence functions with bindings.

5 The new algorithm, the same example

In this section the new algorithm for evaluating (higher order) attribute grammars is explained. The explanation is based upon the example in the previous section and consists of the following three steps. Each of these steps will be discussed in more detail in the next sections.

1. Consider the tree $T =$ **root**(**decl**(**name**(A), **stat**(**name**(A), **empty**()))) which describes the program (`var A; use A;`). The attribute *code* for T can be obtained by calling visit-S-1·T. Instead of visiting tree T we define new constructors **root**, **decl**, **stat**, **empty**, and **name** which are used in constructing visit functions corresponding to trees. These constructors take the visit functions of the subtrees as parameters and

construct the visit functions for the parent tree. Calling the parameterless *visit function* **root**(**decl**(**name**(A), **stat**(**name**(A), **empty**()))) returns the attribute *code* for T.

2. In order to have an efficient incremental construction and evaluation of visit functions, the visit functions constructors will be memoed.

3. During the construction of visit functions the copy rules (identity functions) may be eliminated, resulting in faster functions and more cache hits.

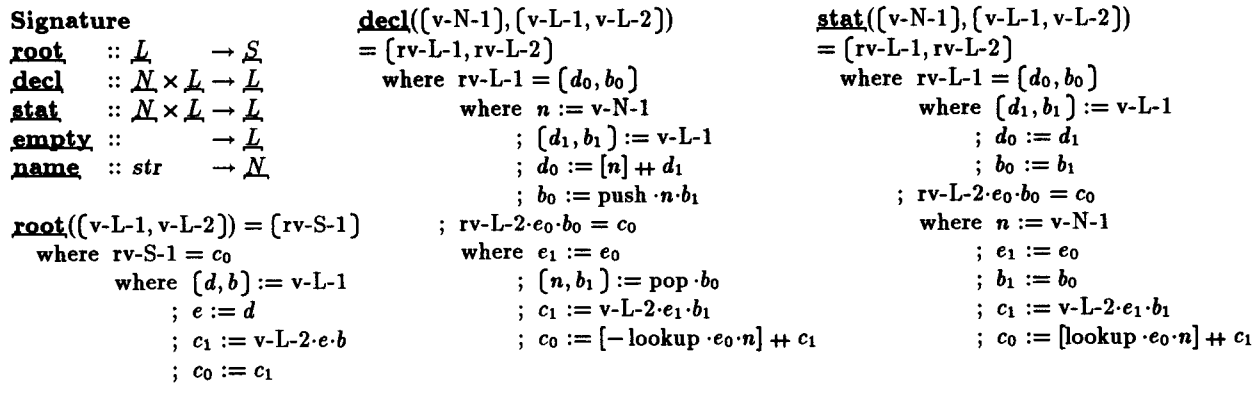

Figure 8: Tuple constructors.

5.1 Removing the tree

How do we "remove" the tree? As stated in the introduction to this section, we are looking, for example, for a structure $\underline{l}$ representing a tree l of type L for which we have the property

$$\underline{l} = [\text{visit-L-1}\cdot l\ ,\ \text{visit-L-2}\cdot l]$$

so that we may call $(\pi_2 \cdot \underline{l})\cdot e\cdot b$ instead of visit-L-2$\cdot l\cdot e\cdot b$ (operator π is the tuple-projection: $\pi_i\cdot[v_1,\ldots,v_n] = v_i$; note also that we use "straight" brackets $[\ldots]$ for tuples). In general a tree is represented by a tuple of its partially parameterised visit functions. We call such a tuple a lifted tree. We denote lifting with the symbol $\underline{\ }$. In the table below we show how to determine the signatures of the lifted trees.

Tree	Function	Signature	Lifted signature	Lifted tree
S	visit-S-1	$S \rightarrow code$	$\underline{S_1} := \rightarrow code$	$\underline{S} := \underline{S_1}$
L	visit-L-1	$L \rightarrow env \times bind$	$\underline{L_1} := \rightarrow env \times bind$	$\underline{L} := \underline{L_1} \times \underline{L_2}$
	visit-L-2	$L \times env \times bind \rightarrow code$	$\underline{L_2} := env \times bind \rightarrow code$	
N	visit-N-1	$N \rightarrow str$	$\underline{N_1} := \rightarrow str$	$\underline{N} := \underline{N_1}$

The last column of the table shows that the type of the $\underline{l}$ tree introduced at the begining of this section is $\underline{L}$ (thus $\underline{l} :: \underline{L}$) so that $\pi_1\cdot\underline{l} :: \underline{L_1}$ and $\pi_2\cdot\underline{l} :: \underline{L_2}$.

How do we compute those lifted trees? We have to lift the original tree constructors for that. For example constructor **stat** $:: N \times L \rightarrow L$ is lifted to constructor **stat** $:: \underline{N} \times \underline{L} \rightarrow \underline{L}$.

Hence, **stat** constructs a two-tuple of visit functions to an L-tree, by combining the visit functions of its sons: an N-tree (represented by a one-tuple $\underline{N}$) and an L-tree (represented by a two-tuple $\underline{L}$). The definition of this construction is given in Figure 8.

5.2 Memoing constructors

Terms are often implemented by means of pointer structures. In order to save space, memoconstructors can be used: equal structures are shared. This has another major advantage: the term equality test reduces to fast pointer comparison. The effect of memoing constructors on the PASCAL-like program fragment

```
x:=x xor y; y:=x xor y; x:=x xor y
```

is illustrated in Figure 9. This figure just illustrates the basic idea. The main advantage of memoing constructors is not the sharing of subtrees within a single tree (for example caused by multiple instances of a non-terminal attribute), but the sharing of subtrees between the old version and the updated version of the tree in an incremental environment.

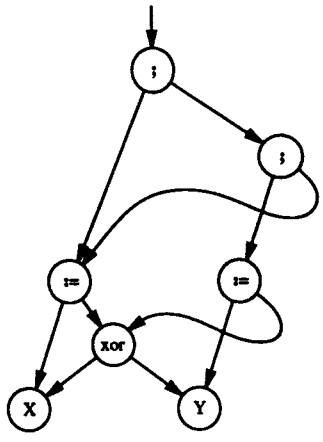

Figure 9: Memoing constructors for statements and expressions.

Memoing is implemented as a "shell" around the constructor functions. Each constructor has its own memoconstructor; for efficiency reasons, the caches (hash tables) may be merged. As an example, see the memoed concat constructor on statements below:

```
function memo-concat(s0 : stat; s1 : stat) : stat
  var s2 : stat
  begin
    if InHashtable("concat", s0, s1)
      then s2 := GetFromHashtable("concat", s0, s1)
      else s2 := concat(s0, s1); InsertInHashtable(s2)
    fi;
    return(s2)
  end
```

Figure 10.**a** represents a sample tree for our grammar. Each circle represents a treenode and the names below it label the tree constructors used. Figure 10.**b** shows an equivalent but lifted

version. Each circle now represents a treenode that encapsulates a tuple of visit functions which, in turn, are represented by the (lines and) solid discs. The names label the tuple constructors used. Although a lifted tree constructor generates a tuple of functions (instead of one term), equal terms in the old case yield equal tuples in the lifted case. Memoing is not spoiled by lifting.

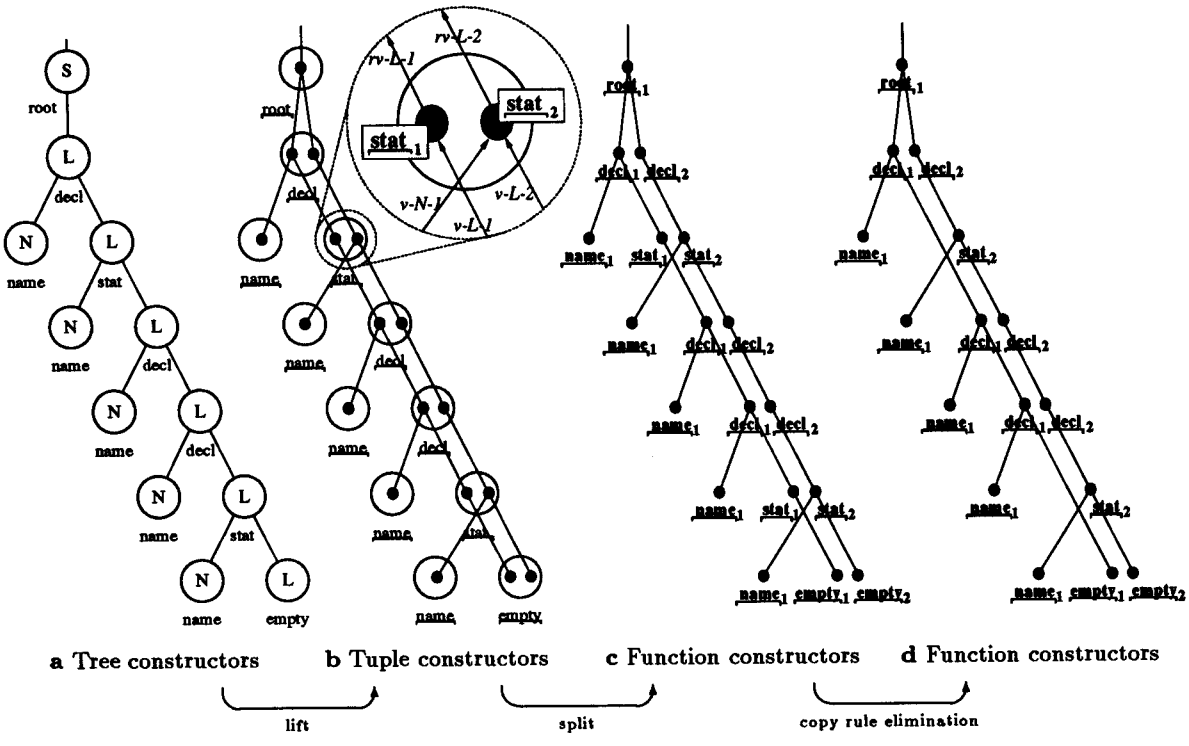

Figure 10: The different constructors.

Our goal, exploiting the independence of separate visits, is not yet reached however. We have to split the tuples into the visit functions they consist of: instead of memoing the tuple constructors we memo the visit function constructors. This is depicted in Figure 10.**c**; solid discs are still visit function nodes, the names label visit function constructors.

The result of this transformation is a better incremental behaviour. Changing the identifier of a **stat**-production imposes no changes on the first visit function because $\mathbf{stat}_1$ has no $\mathbf{name}_1$-son. So S is granted with a cache hit when calling this function. When changing the identifier of a **decl** production the second visit function doesn't change either. Note however that when this visit function is called after the edit action, both its (input)parameters are likely to differ from the previous call, since they both depend on the identifiers declared; so a cache hit is unlikely in this case.

Constructors of visit functions may be thought of as an incomplete data flow graphs. These graphs have holes, where other graphs need to be pasted in. For example, in production **stat** (Figure 8) the graph for the second visit sequence function, rv-L-2, has two holes. One hole for v-N-1 and one for v-L-2. Formulated differently, rv-L-2 = $\underline{\mathbf{stat}}_2$(v-N-1, v-L-2) where visit

function constructor $\underline{\mathbf{stat}}_2$ is the graph of the combinator:

$$\begin{aligned}\underline{\mathbf{stat}}_2(f,g) = [&\lambda \text{ v-N-1} :: \underline{N}_1 \text{ , v-L-2} :: \underline{L}_2 \text{ , } e_0 :: \mathit{env} \text{ , } b_0 :: \mathit{bind} \\ &: c_0 \text{ where } n := \text{v-N-1};\ e_1 := e_0;\ b_1 := b_0;\ c_1 := \text{v-L-2}{\cdot}e_1{\cdot}b_1;\ c_0 := [\text{lookup}{\cdot}e_0{\cdot}n] +\!\!+ c_1 \\ &]{\cdot}f{\cdot}g\end{aligned}$$

Since we have eight function bodies (visit subsequences) we will get eight visit function constructors: $\underline{\mathbf{root}}_1$, $\underline{\mathbf{decl}}_1$, $\underline{\mathbf{decl}}_2$, $\underline{\mathbf{stat}}_1$, $\underline{\mathbf{stat}}_2$, $\underline{\mathbf{empty}}_1$, $\underline{\mathbf{empty}}_2$, and $\underline{\mathbf{name}}_1$. The signatures of these constructors vary depending on the functions they bind. As we saw before, constructor $\underline{\mathbf{stat}}_2$ binds the first (and only) visit to N (typed $\underline{N}_1$) and a second visit to L (typed $\underline{L}_2$). Hence $\underline{\mathbf{stat}}_2 :: \underline{N}_1 \times \underline{L}_2 \to \underline{L}_2$. Likewise we find

$$\begin{aligned}\underline{S}_1 &= \underline{\mathbf{root}}_1(\underline{L}_1, \underline{L}_2) \\ \underline{L}_1 &= \underline{\mathbf{decl}}_1(\underline{N}_1, \underline{L}_1) \\ &\mid \underline{\mathbf{stat}}_1(\underline{L}_1) \\ &\mid \underline{\mathbf{empty}}_1() \\ \underline{L}_2 &= \underline{\mathbf{decl}}_2(\underline{L}_2) \\ &\mid \underline{\mathbf{stat}}_2(\underline{N}_1, \underline{L}_2) \\ &\mid \underline{\mathbf{empty}}_2() \\ \underline{N}_1 &= \underline{\mathbf{name}}_1(\mathit{str})\end{aligned}$$

Another way of looking at it is the following transformation of constructor **stat**.

$$\begin{aligned}& \mathbf{stat} :: N \times L \to L \\ \equiv\ & \quad \{ \text{ lifting } \} \\ & \underline{\mathbf{stat}} :: \underline{N} \times \underline{L} \to \underline{L} \\ \equiv\ & \quad \{ \text{ encapsulation } \} \\ & \underline{\mathbf{stat}}_1 :: \underline{N} \times \underline{L} \to \underline{L}_1 \\ & \underline{\mathbf{stat}}_2 :: \underline{N} \times \underline{L} \to \underline{L}_2 \\ \equiv\ & \quad \{ \text{ abbreviation } \} \\ & \underline{\mathbf{stat}}_1 :: \underline{N}_1 \times (\underline{L}_1 \times \underline{L}_2) \to \underline{L}_1 \\ & \underline{\mathbf{stat}}_2 :: \underline{N}_1 \times (\underline{L}_1 \times \underline{L}_2) \to \underline{L}_2 \\ \equiv\ & \quad \{ \text{ split (each visit function is called exactly once) } \} \\ & \underline{\mathbf{stat}}_1 :: \quad \underline{L}_1 \quad \to \underline{L}_1 \\ & \underline{\mathbf{stat}}_2 :: \underline{N}_1 \times \quad \underline{L}_2 \to \underline{L}_2\end{aligned}$$

The original tree constructors have been replaced by the tuple constructors which encapsulate the visit function constructors. In Figure 11 this is made explicit. These visit function constructors are the "smallest" constructors, hence they form the ideal "grain of memoing".

$$\begin{array}{lll}
\underline{\mathbf{root}} & :: \underline{L} \to \underline{S} & \underline{\mathbf{root}}([\text{v-L-1}, \text{v-L-2}]) = [\underline{\mathbf{root}}_1(\text{v-L-1}, \text{v-L-2})] \\
\underline{\mathbf{decl}} & :: \underline{N} \times \underline{L} \to \underline{L} & \underline{\mathbf{decl}}([\text{v-N-1}], [\text{v-L-1}, \text{v-L-2}]) = [\underline{\mathbf{decl}}_1(\text{v-N-1}, \text{v-L-1}), \underline{\mathbf{decl}}_2(\text{v-L-2})] \\
\underline{\mathbf{stat}} & :: \underline{N} \times \underline{L} \to \underline{L} & \underline{\mathbf{stat}}([\text{v-N-1}], [\text{v-L-1}, \text{v-L-2}]) = [\underline{\mathbf{stat}}_1(\text{v-L-1}), \underline{\mathbf{stat}}_2(\text{v-N-1}, \text{v-L-2})] \\
\underline{\mathbf{empty}} & :: \to \underline{L} & \underline{\mathbf{empty}}() = [\underline{\mathbf{empty}}_1(), \underline{\mathbf{empty}}_2()] \\
\underline{\mathbf{name}} & :: \mathit{str} \to \underline{N} & \underline{\mathbf{name}}(i) = [\underline{\mathbf{name}}_1(i)]
\end{array}$$

Figure 11: Tuple constructors encapsulating the visit function constructors.

5.3 Removing copy rules

In the previous section we noticed that a change to an N-son of a **stat** node doesn't change the first visit function: a **stat** production adds nothing to the first visit. It inserts the identity function; the functional equivalent of copy rules.

The final improvement we achieve is the elimination of such copy rules. In our example, visit function constructor $\underline{\textbf{stat}}_1$ is such a copy rule. To see this, we derive

$$
\begin{array}{ll}
 & \underline{\textbf{stat}}_1(f) \\
= & \quad \{ \text{ see Figure 8 } \} \\
 & [\lambda \text{ v-L-1 } :: \mathcal{L}_1 \\
 & : (d_0, b_0) \text{ where } (d_1, b_1) := \text{v-L-1};\ d_0 := d_1;\ b_0 := b_1 \\
 &]\cdot f \\
= & \quad \{ \text{ pairing } \} \\
 & [\lambda \text{ v-L-1 } :: \mathcal{L}_1 \\
 & : (d_0, b_0) \text{ where } (d_1, b_1) := \text{v-L-1};\ (d_0, b_0) := (d_1, b_1) \\
 &]\cdot f \\
= & \quad \{ \text{ substitution } \} \\
 & [\lambda \text{ v-L-1 } :: \mathcal{L}_1 \\
 & : (d_0, b_0) \text{ where } (d_0, b_0) := \text{v-L-1} \\
 &]\cdot f \\
= & \quad \{ \text{ substitution } \} \\
 & [\lambda \text{ v-L-1 } :: \mathcal{L}_1 \\
 & : \text{v-L-1} \\
 &]\cdot f \\
= & \quad \{ \beta\text{-reduction } \} \\
 & f
\end{array}
$$

Since this constructor is nothing but the identity constructor, tuple constructor $\underline{\textbf{stat}}$ may be changed to:

$$\underline{\textbf{stat}}((\text{v-N-1}), (\text{v-L-1}, \text{v-L-2})) = (\text{v-L-1}, \underline{\textbf{stat}}_2(\text{v-N-1}, \text{v-L-2}))$$

incorporating the elimination of copy rules. Figure 10.**d** shows the resulting tree. We now not only have cache hits for the first visit function when we *change* the identifier of a **stat** node; we even have cache hits for the first visit function after *adding* new **stat** productions! And in addition to this speed up, we also gain in memory usage.

6 Conclusions

We have presented a new evaluation technique for attribute grammars based on a functional approach. One large evaluation function is constructed for a given tree. This evaluation function can be efficiently incrementally updated using memoed constructors for visit functions. Efficient incremental evaluation is achieved by caching these functions. Furthermore, since visit functions now only depend on that part of the tree which is actually visited by them, more cache hits are expected. A second advantage of our approach is that copy rules may be removed automatically, thus saving visits and maximizing cache hits.

References

[Al91] Alblas, Henk. *Introduction to Attribute Grammars.* In H. Alblas and B. Melichar (Eds.) *Attribute Grammars, Applications and Systems (SAGA '91)*, Lecture Notes in Computer Science, Vol. **545**, pages 1–15, Springer-Verlag, June 1991.

[DeJoLo88] Deransart, Pierre, Martin Jourdan and Bernard Lorho. *Attribute Grammars. Definitions, Systems and Bibliography.* Lecture Notes in Computer Science, Vol. **323**, Springer-Verlag, August 1988.

[DeJo90] Deransart, P, M Jourdan (Eds.). *Attribute Grammars and their Applications (WAGA '90).* Lecture Notes in Computer Science, Vol. **461**, Springer-Verlag, September 1990.

[Hu85] Hughes, John. *Lazy memo-functions.* In Jean-Pierre Jouannaud (Ed.) *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, Vol. **201**, pages 129–146, Springer-verlag, 1985.

[Ka80] Kastens, Uwe. *Ordered Attributed Grammars.* In Acta Informatica, Vol. **13**, pages 229–256, 1980.

[Kn68] Knuth, Donald E. *Semantics of context-free languages.* In Mathematical Systems Theory, Vol. **2**, No. 2, pages 127–145, Springer-Verlag, 1968.

[Kn71] Knuth, Donald E. *Semantics of context-free languages (correction).* In Mathematical Systems Theory, Vol. **5**, No. 1, pages 95–96, Springer-Verlag, 1971.

[Pu88] Pugh, W.W. *Incremental Computation and the Incremental Evaluation of Functional Programs.* Technical Report 88-936 and Ph.D. Thesis, Department of Computer Science, Cornell University, Ithaca, N.Y., August 1988.

[SwVo91] Swierstra, S.D. and H.H. Vogt. *Higher Order Attribute Grammars.* In H. Alblas and B. Melichar (Eds.) *Attribute Grammars, Applications and Systems (SAGA '91)*, Lecture Notes in Computer Science, Vol. **545**, pages 256–296, Springer-Verlag, June 1991.

[TeCh90] Teitelbaum, Tim and Richard Chapman. *Higher-Order Attribute Grammars and Editing Environments.* In Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, Vol. **25**, No. 6 (proceedings), pages 197-208, June 1990.

[Tu85] Turner, D.A. *Miranda: A non-strict functional language with polymorphic types.* In Jean-Pierre Jouannaud (Ed.) *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, Vol. **201**, pages 1-16, Springer-Verlag, 1985.

[VoSwKu89] Vogt, H.H., S.D. Swierstra and M.F. Kuiper. *Higher Order Attribute Grammars.* In Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, Vol. **24**, No. 7 (proceedings), pages 131-145, June 1989.

[VoSwKu91] Vogt, Harald, Doaitse Swierstra and Matthijs Kuiper. *Efficient Incremental Evaluation of Higher Order Attribute Grammars.* In J. Maluszyński and M. Wirsing (Eds.) *Programming Language Implementation and Logic Programming (PLILP '91)*, Lecture Notes in Computer Science, Vol. **528**, pages 231–242, Springer-Verlag, 1991.

[Ye83] Yeh, D. *On incremental evaluation of ordered attributed grammars.* In BIT, Vol. **23**, pages 308-320, 1983.