On the efficient incremental evaluation of Higher Order Attribute Grammars

H.H. Vogt, S.D. Swierstra, M.F. Kuiper

Technical Report RUU-CS-90-36
December 1991
(revised version)

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

ISSN: 0924-3275

On the efficient incremental evaluation of Higher Order Attribute Grammars[*]

Harald Vogt[†], Doaitse Swierstra and Matthijs Kuiper

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

Abstract

This paper presents a new algorithm for the incremental evaluation of Ordered Attribute Grammars (OAGs), which also solves the problem of the incremental evaluation of Ordered *Higher-order* Attribute Grammars (OHAGs). Two new approaches are used in the algorithm.

First, instead of caching all results of semantic functions in the grammar, all results of visits to trees are cached. There are no attributed trees, because all attributes are stored in the cache. Trees are built using hash consing, thus sharing multiple instances of the same tree and avoiding repeated attributions of the same tree with the same inherited attributes. Second, each visit computes not only synthesized attributes but also *bindings* for future visits. Bindings, which contain attribute values computed in one visit and used in future visits, are also stored in the cache. As a result, future visits get the necessary earlier computed values (the bindings) as a parameter.

The algorithm runs in $O(|\mathit{Affected}| + |\mathit{paths\_to\_roots}|)$ steps after modifying subtrees, where $|\mathit{paths\_to\_roots}|$ is the sum of the lengths of all paths from the root to all modified subtrees, which is almost as good as an optimal algorithm for first-order AGs, which runs in $O(|\mathit{Affected}|)$.

1 Introduction

Attribute grammars describe the computation of attributes: values attached to nodes of a tree. The tree is described with a context-free grammar. Attribute computation is defined by semantic functions. AGs are used to define languages and form the basis of compilers, language-based editors and other language-based tools. For an introduction and more on AGs see: [Deransart, Jourdan and Lorho 88, Deransart and Jourdan 90].

[*]Revised version of RUU-CS-90-36 and a copy of the paper in the CSN 91 proceedings.

[†]Support has been received from the Netherlands Organisation for Scientific Research N.W.O. under NFI-project STOP, project Specification and Transformations Of Programs.

Higher-order AGs ([Vogt, Swierstra and Kuiper 89, Teitelbaum and Chapman 90] and [Swierstra and Vogt 91]) 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: First, trees can be *used* directly as a value within an attribute equation. Second, a part of the tree can be *defined* by attribution. Trees used as a value and trees defined by attribution are known as non-terminal attributes (NTAs).

It is known that the (incremental) attribute evaluator for Ordered AGs [Kastens 80, Yeh 83, Reps and Teitelbaum 88] can be trivially adapted to handle Ordered Higher-order AGs [Vogt, Swierstra and Kuiper 89]. The adapted evaluator, however, attributes each instance of equal NTAs separately. This leads to nonoptimal incremental behaviour after a change to a NTA, as can be seen in the recently published algorithm of [Teitelbaum and Chapman 90]. Our evaluation algorithm handles multiple occurrences of the same NTA (and the same subtree) efficiently in $O(|\mathit{Affected}| + |\mathit{paths\_to\_roots}|)$ steps, where $|\mathit{paths\_to\_roots}|$ is the sum of the lengths of all paths from the root to roots of modified subtrees.

The new incremental evaluator can be used for language-based editors like those generated by the Synthesizer Generator [Reps and Teitelbaum 88] and for minimizing the amount of work for restoring semantic values in tree-based program transformations [Vogt, v.d. Berg and Freije 90]. The new algorithm is based on the combination of the following four ideas:

- The algorithm computes attribute values by using visit functions. A visit function takes as first parameter a tree and some of the inherited attributes of the root of the tree. Some of the returned values are synthesized attributes of the root of the tree. Our evaluator consists of visit functions that recursively call each other to attribute the tree.

- As in [Teitelbaum and Chapman 90]'s algorithm, trees are built using hash consing for trees. In our algorithm this is the only representation for trees, thus multiple instances of the same tree will be shared.

  Because many instantiations of a NTA may exists, each with its own attributes, attributes are no longer stored within the tree, but in a cache. In a normal incremental treewalk evaluator a partially attributed tree can be considered as a cache. Attributes needed by a visit and computed by a previous visit are not recomputed but are found in the partially attributed tree.

- A call to a visit function corresponds to a visit in a visit sequence of an Ordered HAG. Instead of caching the results of *semantic functions*, as was done in [Pugh 88], the results of *visit functions* are cached. This is more efficient because a cache hit of a visit function means that this visit to (a possible large) tree can be skipped. Furthermore, a visit function may return the results of several semantic functions at a time.

- Although the above idea seems appealing at first sight, a complication is the fact that attributes computed in an earlier visit may have to be available for later visits.

Bindings contain attribute values computed in one visit and used in future visits to the same tree: each visit function computes synthesized attributes *and* bindings for future visits. Bindings computed by earlier visits are passed as an extra parameter to visit functions.

The visit functions can be implemented in any imperative or functional language. Furthermore, visit functions have no free variables and can therefore be viewed as supercombinators [Hughes 82].

Efficient caching is partly achieved by efficient equality testing between parameters of visit functions, which are trees, inherited attributes and bindings. Therefore, hash consing for trees and bindings is used, so testing for equality between trees and between bindings is reduced to a fast pointer comparison. Furthermore, several space optimizations for bindings are possible.

Although the computation of these bindings may appear to be cumbersome, they have a considerable advantage when evaluating incrementally. They contain precisely the information on which visits depend and no more.

The remainder of this article is structured as follows. Section 2 presents an informal definition and example. Section 3 defines visit functions and bindings. A space optimization for bindings is presented in section 4. We discuss visit function caching in section 5. Section 6 presents the results of a simulation of our algorithm for incremental evaluation for a larger example. The incremental behaviour of the algorithm is discussed in section 7. Section 8 presents the conclusions.

2 Informal definition and example

First, visit sequences from which the visit functions will be derived are presented and illustrated by an example. Then bindings and visit functions for the example will be shown. Finally, incremental evaluation will be discussed.

2.1 Visit (sub)sequences

In [Vogt, Swierstra and Kuiper 89] the equivalent of OAGs [Kastens 80] for HAGs, the so called Ordered Higher-order Attribute Grammars (OHAGs), are defined. An OHAG is characterized by the existence of a total order on the defining attribute occurrences for each production p. This order induces a fixed sequence of computation for the defining attribute occurrences, applicable in any tree production p occurs in. Such a fixed sequence is called a *visit sequence* and will be denoted by $VS(p)$. The following introduction to visit sequences for a HAG is almost literally taken from [Kastens 80].

This evaluation order is the starting point for the construction of a flexible and efficient attribute evaluation algorithm based on visit sequences. It is closely adapted to the particular attribute dependencies of the AG. The principle is demonstrated here. Assume that an instance of X is derived by

$$S \Rightarrow Y \rightarrow_p uXy \rightarrow_q uvwxy \Rightarrow s.$$

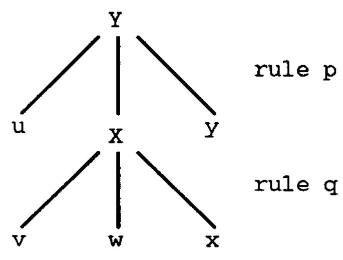


Figure 1:
A structure tree.

The corresponding part of the structure tree is shown in Figure 1. An attribute evaluation algorithm traverses the structure tree using the operations "move down to a descendant node" (e.g. from Y to X) or "move up to the ancestor node" (e.g. from X to Y). During a visit to node Y some attributes defined in production p are evaluated according to semantic functions, if p is applied at Y. In general several visits to each node are needed before all attributes are evaluated. A local tree walk rule (a visit sequence) is associated with each p. It is a sequence of four types of instructions: move up to the ancestor, move down to a certain descendant, evaluate a certain attribute and evaluate followed by expansion of the labeled tree by the value of a non-terminal attribute. The last instruction is specific for a HAG.

VS(p) is split into *visit subsequences VSS(p,v)* by splitting after each "move up to the ancestor" instruction in *VS(p)*. The attribute grammar in Figure 2 is used in the sequel only to demonstrate visit subsequences, bindings and visit functions. Section 6 contains a larger example.

2.2 Visit functions for the example grammar

The evaluator is obtained by translating each visit subsequence *VSS(p,v)* into a *visit function visit_N_v* where N is the left hand side of p.

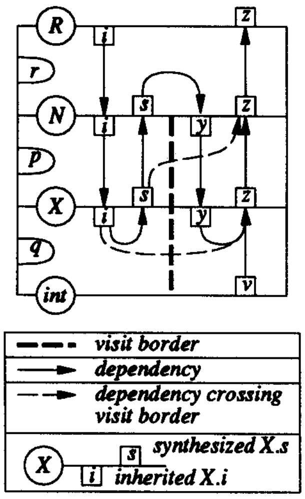
All visit functions together form a functional attribute evaluator program. We use a Miranda-like notation [Turner 1985] for visit functions. Because the visit functions are strict, which results in explicit scheduling of the computation, visit functions could also be easily translated into Pascal or any other non-lazy imperative language.

The first parameter in the definition of *visit_N_v* is a *pattern* describing the subtree to which this visit is applied. The first element of the pattern is a marker, a constant which indicates the applied production rule. The other elements are identifiers representing the subtrees of the node. Following the functional style we will have one set of visit functions for each production with left hand side N.

All other arguments, *except* the last, of *visit_N_v* represent the inherited attributes used in *VSS(p,v)*. Before we discuss the results of a visit function, consider the grammar in Figure 2 again. The inherited attribute $X.i$ and the synthesized attribute $X.s$ in Figure 2 are also used in the second visit to N and X but passed to or computed in the first visit.

Therefore, every *visit_N_v* not only computes synthesized attributes but also *bindings* (inherited and synthesized attributes *computed* in *visit_N_v* and *used* in subsequent visits

$r{:}R \rightarrow N \quad \{\ N.i := R.i;\ N.y := N.s;\ R.z := N.z;\ \}$
$p{:}N \rightarrow X \quad \{\ X.i := N.i;\ N.s := X.s;\ X.y := N.y;\ N.z := X.z + X.s;\ \}$
$q{:}X \rightarrow int \quad \{\ X.s := X.i;\ X.z := X.y + X.i + int.v;\ \}$



```
VS(p)                          VS(q)
 = VSS(p,1)                     = VSS(q,1)
   = X.i := N.i                   = X.s := X.i
   ; Visit X,1                    ; VisitParent 1
   ; N.s := X.s                 ; VSS(q,2)
   ; VisitParent 1                = X.z := X.y + X.i + int.v
 ; VSS(p,2)                       ; VisitParent 2
   = X.y := N.y
   ; Visit X,2
   ; N.z := X.z + X.s
   ; VisitParent 2
```

Figure 2: An example AG (top), the dependencies (left) and visit sequences (right). The dashed lines indicate dependencies of an attribute computed in the second visit on an attribute defined in the first visit. *VS(r)* is omitted.

to N). So *visit_N_v* computes a list of $(nov_N\text{-}v)$ bindings, one for each subsequent visit (here nov_N is the number of visits to N). The bindings *used* in *visit_N_v+i* but *computed* in *visit_N_v* are denoted by $binds\_N^{v \rightarrow v+i}$.

The last argument of *visit_N_v* is a list of bindings for *visit_N_v* computed in earlier visits $1 \ldots (v-1)$ to N. The bindings themself are lists containing attribute values and further bindings. Both lists are constructed using hash consing. Elements of a list are addressed by projection, e.g. $binds\_N^{i \rightarrow v}.1$ is the first element of the list.

We now turn to the visit functions for the visit subsequences *VSS(p,v)* and *VSS(q,v)* of grammar in Figure 2. We will put a box around attributes that are returned in a binding. In the example this concerns $\boxed{X.i}$ and $\boxed{X.s}$. The first visit to N will return the synthesized attribute $N.s$, and a binding list $binds\_N^{1 \rightarrow 2}$ containing the later needed $X.s$ together with $binds\_X^{1 \rightarrow 2}$. The binding list $binds\_N^{1 \rightarrow 2}$ is defined by $[\boxed{X.s}, binds\_X^{1 \rightarrow 2}]$.

```
visit_N_1 (p [X]) N.i = (N.s, binds_N^{1→2})
        where X.i = N.i
              (X.s, binds_X^{1→2}) = visit_X_1 X X.i
              N.s = X.s
              binds_N^{1→2} = [ [X.s] , binds_X^{1→2} ]
```

In the above definition $(p\ [X])$ denotes the first argument: a tree at which production p is applied, with one son, X. The second argument is the inherited attribute i of N. The function returns the synthesized attribute s and binding for the second visit to N ($X.s$ together with the bindings from the first visit to subtree X). Function *visit_N_2* does not return a binding because it is the *last* visit to a N-tree.

```
$visit\_N\_2\ (p\ [X])\ N.y\ [binds\_N^{1\to 2}] = N.z$
        where $X.y = N.y$
              $binds\_X^{1\to 2} = binds\_N^{1\to 2}.2$
              $X.z = visit\_X\_2\ X\ X.y\ [binds\_X^{1\to 2}]$
              $\boxed{X.s} = binds\_N^{1\to 2}.1$
              $N.z = X.z + X.s$
```

The other visit functions have a similar structure.

```
$visit\_X\_1\ (q\ [int])\ X.i = (X.s,\ binds\_X^{1\to 2})$
        where $X.s = X.i$
              $binds\_X^{1\to 2} = [\boxed{X.i}]$

$visit\_X\_2\ (q\ [int])\ X.y\ [binds\_X^{1\to 2}] = X.z$
        where $\boxed{X.i} = binds\_X^{1\to 2}.1$
              $X.z = X.y + X.i + int.v$
```

We have chosen the order of definition and use in the `where` clause in such a way that the visit functions could be also defined in an imperative language. A `where` clause contains three kinds of definitions:

1. assignments and visits from the corresponding *VSS(p,v)*.
2. lookups of attributes and bindings in bindings (for example in *visit_N_2* the binding $binds\_X^{1\to 2}$ is looked up in $binds\_N^{1\to 2}$).
3. definitions for returned bindings. The precise definition of visit functions and bindings is given in section 3.

2.3 Incremental evaluation

After a tree T is modified into *T'*, in our model with hash consed trees, *T'* shares all unmodified parts with T. To evaluate the attributes of T and *T'* the *same* visit function *visit_R_1* is used, where R is the root non-terminal. Note that tree *T'* is totally rebuilt before *visit_R_1* is called.

The incremental evaluator automatically skips unchanged parts of the tree because of cache-hits of visit functions, provided the inherited attributes have not changed. Hash consing for trees and bindings is used to achieve efficient caching, for which fast equality tests are essential. Separate bindings for each visit are computed, so for example *visit_N_1* and *visit_N_4* could be recomputed after a subtree replacement, but *visit_N_{2,3}* could be found in the cache and skipped. Some other advantages are illustrated in Figure 3, in which the following can be noted:

- *NTA1* and *NTA2* are defined by attribution, indicated by boxes (□).
- Multiple instances of the same (sub)tree, for example multiple instantiated NTAs, are *shared* by using hash consing for trees (Trees *T2* and *T'2*).
- Those parts of an attributed tree derived from *NTA1* and *NTA2* which can be reused after *NTA1* and *NTA2* change value are *identified automatically* because of the hash consing for trees and cached visit functions (Trees *T3* and *T4* in (b)). This holds also for a subtree modification in the initial parse tree (Tree *T1*).
- Because trees *T1*, *T3* and *T4* may be attributed the same in *(a)* and *(b)* they will be skipped after the subtree modification and the amount of work which has to be done in (b) is $O(|\mathit{Affected}\ T'2| + |\mathit{paths\_to\_roots}|)$ steps, where $|\mathit{paths\_to\_roots}|$ is the sum of the lengths of all paths from the root to all subtree modifications (*NEW*, *X1* and *X2*).

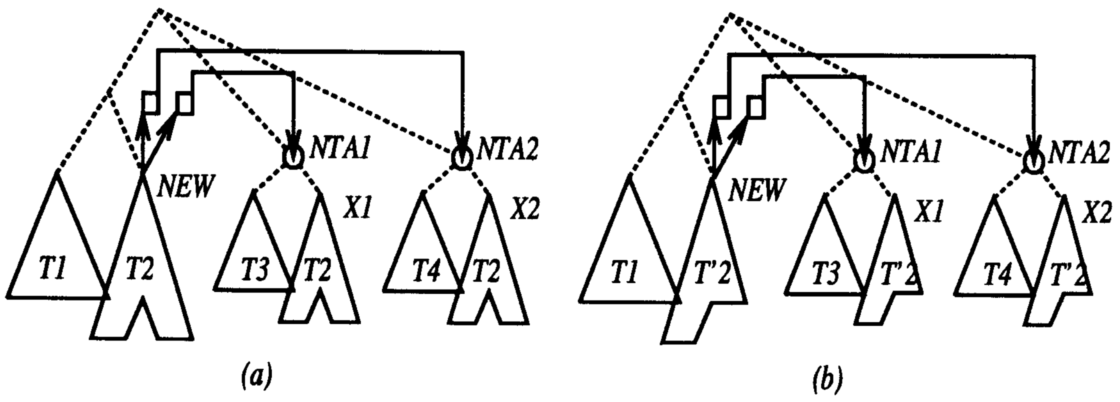


Figure 3: A subtree modification at node *NEW* induces subtree modifications at nodes *X1* and *X2* in the trees derived from *NTA1* and *NTA2*. In this example we suppose that all instantiated trees *T1*, *T2*, *T'2*, *T3* and *T4* are attributed the same in *(a)* and *(b)*.

3 Visit functions and bindings

We now turn to the definition of visit functions and bindings.

Let p be a production of the form $p{:}N \rightarrow \ldots X_i \ldots$. Let *VS(p)* be the visit sequence for p. Let nov_N be the number of visits to N. Let $VSS(p,1) \ldots VSS(p,nov_N)$ be the visit subsequences in *VS(p)*.

VSS(p,v) is translated into the visit function *visit_N_v* as follows:

$$\begin{aligned} &\mathit{visit\_N\_v}\ (p\ [\ldots X_i \ldots])\ \mathit{inh}_v^N\ [\mathit{binds\_N}^{1\rightarrow v}, \ldots, \mathit{binds\_N}^{(v-1)\rightarrow v}] = \\ &\qquad (\mathit{syn}_v^N, \mathit{binds\_N}^{v\rightarrow v+1}, \ldots, \mathit{binds\_N}^{v\rightarrow nov_N}) \\ &\qquad \texttt{where}\ \mathit{Lines\ from\ 1)\ to\ 3)}. \end{aligned}$$

1) The assignments and visits in VSS(p,v).
2) Lookups of attributes and bindings computed in earlier visits.
3) Definitions for the returned bindings.

inh_v^N are the available inherited attributes needed in and not available in visits before *VSS(p,v)*. syn_v^N are the synthesized attributes computed in *VSS(p,v)*. The elements 1) to 3) are defined as follows. 1) is just copying from *VSS(p,v)*. In 1) a *Visit* X_i, w is translated into

$$(syn_w^X,\ binds\_X^{w \to w+1},\ \ldots,\ binds\_X^{w \to novx}) = \\ visit\_X\_w\ X_i\ inh_w^X\ [binds\_X^{1 \to w},\ \ldots,\ binds\_X^{(w-1) \to w}]$$

When X_i is a non-terminal attribute, the variable defining X_i is used as the first argument pattern for $visit\_X_i\_w$.

There are three kinds of lookups in 2): Inherited attributes, synthesized attributes and bindings. The lookup method is the same for all, so we will only describe the method for an inherited attribute here. Let *N.inh* be an inherited attribute of *N* which is used in *visit_N_v* but *not* defined in *visit_N_v*. Then, the lookup $N.inh = binds\_N^{e \to v}.f$ is added, for the appropriate e and f.

In 3) the bindings results of *visit_N_v* are defined. Recall that the $binds\_N^{v \to v+i}$ are defined in terms of the visit sequence of production p. $binds\_N^{v \to v+i}$ is defined as a list containing those inherited attributes of *N* and synthesized attributes of sons of *N* used in *visit_N_v* *and* in *visit_N_v+i* (denoted by ${inout\_N_p}^{v \to v+i}$) **plus** the bindings computed by visits to *N*'s sons during *visit_N_v* which are used in future visits to those sons during *visit_N_v+i* (denoted by ${binds\text{-}sons\_N_p}^{v \to v+i}$). For example $binds\_N^{1 \to 2}$ in the example visit functions in section 2 is

$$binds\_N^{1 \to 2} = [{inout\_N_p}^{1 \to 2},\ {binds\text{-}sons\_N_p}^{1 \to 2}] = [X.s,\ binds\_X^{1 \to 2}],$$

where during execution the value of $binds\_N^{1 \to 2}$ will be $[X.s,\ [X.i]]$. ${inout\_N_p}^{v \to v+i}$ and ${binds\text{-}sons\_N_p}^{v \to v+i}$ are defined as follows:

$${inout\_N_p}^{v \to v+i} = (N.inh \cup X.syn) \cap VSS(p,v) \cap VSS(p,v+i)$$

$${binds\text{-}sons\_N_p}^{v \to v+i} = \{\ binds\_X^{w \to j} \mid (visit\_X\_w \in VSS(p,v)) \\ \wedge (visit\_X\_j \in VSS(p,v+i))\ \}$$

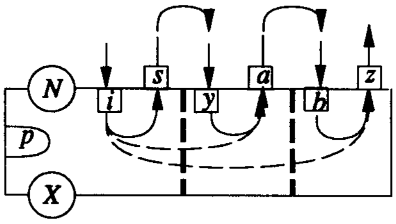
The following theorem holds for the above defined functional program.

Theorem 3.1 *Let HAG be a well-defined Ordered Higher Order Attribute grammar, and let S be a structure tree of HAG. The execution of the above defined functional program for HAG with input S terminates and attributes the tree S correctly. Furthermore, no attributes are evaluated twice.*

4 Optimizations for bindings

Several optimizations that reduce the number of occurrences of values in $binds\_N^{v \to v+i}$ are possible.

We discuss three optimizations that reduce the number of occurrences of attribute values in $binds\_N^{v \to i}$ and one optimization that does the same for the binding list. The first optimization reduces double occurrences of attribute values in different bindings and is illustrated by the following production



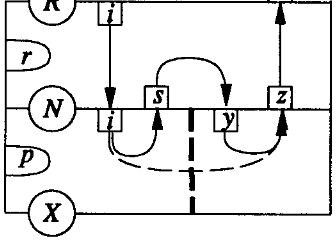
Here
$binds\_N^{1 \to 2} = binds\_N^{1 \to 3} = [N.i]$
and
$binds\_N^{2 \to 3} = [N.i]$

The $binds\_N^{1 \to 3}$ can be omitted by only binding $N.i$ for the *first* visit which uses $N.i$ after *visit_N_1*. This can be achieved by defining $inout\_N_p{}^{v \to v+i}$ in $binds\_N^{v \to v+i}$ as follows

$$
\begin{aligned}
inout\_N_p{}^{v \to v+i} = \{N.inh \;&|\; N.inh \in VSS(p,v) \\
&\wedge\; N.inh \in VSS(p,v+i) \\
&\wedge\; \neg(\exists j \;:\; v < j < v+i \;:\; N.inh \in VSS(p,j))\ \} \\
\cup \;\;& \textit{the same extension for } X.syn \textit{ of course}
\end{aligned}
$$

In words, $binds\_N^{v \to v+i}$ (the bindings defined in *visit_N_v* and needed in *visit_N_v+i*) contains only *N.inh* if *visit_N_v+i* is the *first* visit after *visit_N_v* using *N.inh*. This construction eliminates $binds\_N^{1 \to 3} = [N.i]$ from the example.

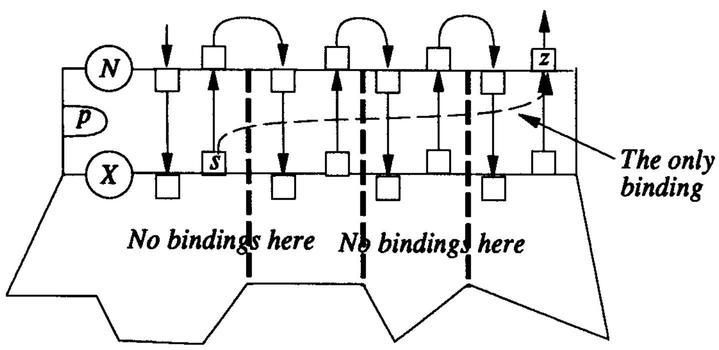
The second optimization reduces inherited attributes from bindings, as is illustrated by the following example



Here $binds\_N^{1 \to 2} = [N.i]$
and $VS(r) =$ *Eval N.i*
Visit N,1
Eval N.y
Visit N,2
Eval R.z
VisitParent 1

Note that *VS(r)* is mapped into *one* visit function *visit_K_1*. Here *N.i* is bound, and also stored local to *visit_K_1* since the two visits to *N* occur in the *same* visit function *visit_K_1*! So *N.i* can be passed directly as an argument to the second visit to *N* and can be removed from $binds\_N^{1 \to 2}$.

The third optimization removes empty bindings from the visit functions and is explained by the following example



Here
$binds\_N^{1\rightarrow 4} = [X.s]$
and
$binds\_N^{1\rightarrow\{2,3\}} = \emptyset$

Some $binds\_N^{v\rightarrow v+i}$ may be always empty. Whether a binding will be always empty can be deduced statically from the attribute grammar.

The last optimization reduces the space for hash consed binding lists. Hash consing lists in the "right order" omits duplicate attribute values. For example, if $binds\_N_p{}^{v\rightarrow v+i} = [C.z,$ $C.w,\ C.y,\ B.a,\ bsons\_C,\ bsons\_B]$ and $binds\_N_s{}^{v\rightarrow v+i} = [C.z,\ B.a,\ C.y,\ bsons\_C]$ then they should be consed as $[C.w,\ bsons\_B,\ \boxed{C.z,\ C.y,\ B.a,\ bsons\_C}]$ and $[\boxed{C.z,\ C.y,\ B.a,\ bsons\_C}]$.
Instead of consing individual elements of a list, parts of a list can be sometimes consed as a whole. In the above example $\boxed{C.z,\ C.y,\ B.a,\ bsons\_C}$ could be consed as one record, thus saving pointer space.

5 Visit function caching

This section describes the implementation of function caching used for caching the visit functions of the functional evaluator and was inspired upon [Pugh 88]. A *hash table* is used to implement the cache. A single cache is used to store the cache results for all functions. Tree T, labeled with root N, is attributed in visit v by calling

```
visit_N_v T inherited_attributes bindings
```

The result of this function is uniquely determined by the function-name, the arguments of the function and the bindings. The visit functions can be cached as follows:

```
cached_apply(visit_N_v, T, inhs, binds) =
    index := hash(visit_N_v, T, inhs, binds)
    ∀ <function, tree, arguments, bindings, result> ∈ cache[index] do
        if function = visit_N_v and EQUAL(tree,T)
           and EQUAL(arguments,inhs) and EQUAL(bindings,binds)
        then return result
    result := visit_N_v T inhs binds
    cache[index] := cache[index] ∪ {<visit_N_v, T, inhs, binds, result>}
    return result
```

To implement visit function caching, we need efficient solutions to several problems. We need to be able to

- compute a hash index based on a function name and an argument list. For a discussion of this problem, see [Pugh 88] for more details.
- determine whether a pending function call matches a cache entry, which requires efficient testing for equality between the arguments (in case of trees and bindings very large structures!) in the pending function call and in a candidate match.

The case of trees and bindings in the last problem is solved by hash consing for trees and bindings.

6 A large example

Consider the HAG in Figure 4, which describes the mapping of a structure consisting of a sequence of defining identifier occurrences and a sequence of applied identifier occurrences onto a sequence of integers containing the index positions of the applied occurrences in the defining sequence. For example, the sentence **let** *a,b,c* **in** *c,c,b,c* **ni** is mapped onto the sequence [*3,3,2,3*].

block(_,_):	*ROOT*	$\rightarrow$	let *DECLS* in *APPS* ni
def(_,_):	*DECLS*	$\rightarrow$	*DECLS ident*
empty_decls():	*DECLS*	$\rightarrow$	*EMPTY_decls*
use(_,_):	*APPS*	$\rightarrow$	*APPS ident* $\overline{ENV}$
			$\overline{ENV} := APPS_0.env$
			$\overline{ENV}.param := ident.id$
			$APPS_1.env := APPS_0.env$
			$APPS_0.seq := APPS_1.seq$ ++ $[\overline{ENV}.index]$
empty_use():	*APPS*	$\rightarrow$	*EMPTY_apps*
update(_,_,_):	*ENV*	$\rightarrow$	*ident number ENV*
empty_env():	*ENV*	$\rightarrow$	*EMPTY_env*

Figure 4: The higher-order AG, only the attribution rules for *use()* are shown.

The following can be noted in Figure 4:

- The attribute $\overline{ENV}$ in *use(_,_)* is a non-terminal (higher-order) attribute. The tree structure is built using the *constructor functions update(_,_,_)* and *empty_env()*, which correspond to the respective productions for $\overline{ENV}$. The attribute $\overline{ENV}$ is instantiated (i.e. a copy of the tree is attributed) in the occurrence of the first production of *APPS*, and takes the rôle of a semantic function.
- Note that there may exist many instantiations of the $\overline{ENV}$-tree, some with different attributes. There thus does not any longer exist an one-to-one correspondence between attributes and abstract-syntax trees.

Figure 5.*a* shows the tree for the sentence **let** *a,b,c* **in** *c,c,b,c* **ni** which was attributed by a call to

visit_ROOT_1 (block(def(def(def(def empty_decls a) b) c))
(use(use(use(use(use empty_use c) c) b) c)))

Incremental reevaluation after removing the declaration of c is done by calling

visit_ROOT_1 (block(def(def(def empty_decls a) b))
(use(use(use(use(use empty_apps c) c) b) c)))

The resulting tree is shown in Figure 5.*b*, note that only the *APPS*-tree will be totally revisited, the first visits to the *DECLS* and *ENV* trees generate cache-hits and further visits to them are skipped. Simulation shows that in this example 75% of all visit-function calls and tree-build calls which have to be computed in 5.*b* if there was no cache, are found in the cache (built up by 5.*a*) when using caching. So 75% of the "work" was saved!

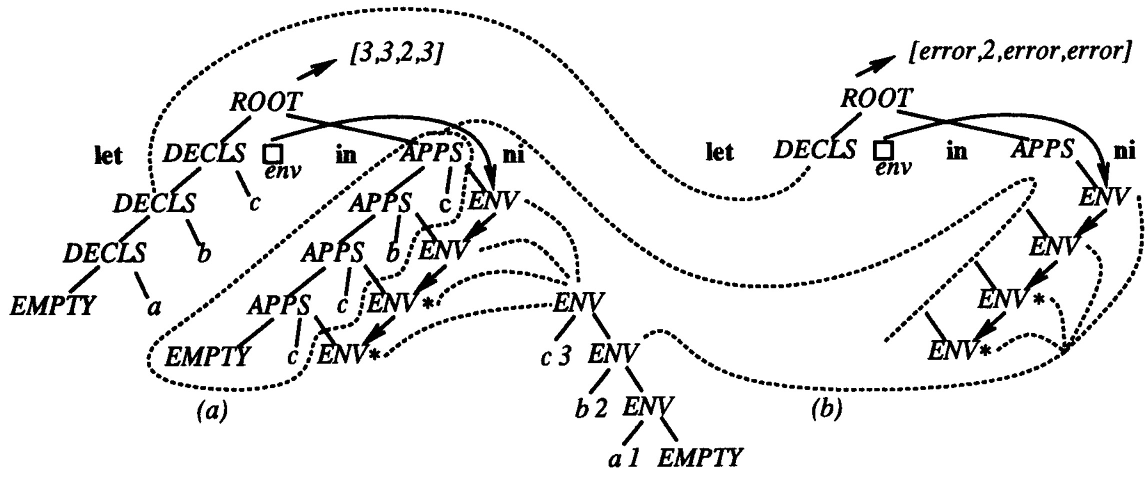


Figure 5: The tree before *(a)* and after *(b)* removing *c* from the declarations in **let** *a,b,c* **in** *c,c,b,c* **ni**. The * indicate cache-hits looking up *c*. The dashed lines denote sharing.

7 Incremental evaluation performance

In this section the performance of the functional evaluator with respect to incremental evaluation is discussed. We would like to prove that the derived incremental evaluator recomputes at most $O(|Affected|)$ attributes. Here *Affected* is the set of attribute instances in the tree which contain a different value after a subtree modification, together with the set of attribute instances newly created.

This desire can be only partly fulfilled; it will be shown that the worst case boundary is given by $O(|Affected| + |paths\_to\_roots|)$. Here *paths_to_roots* is the set of all nodes on the path to the initial subtree modification and the nodes on the paths to the root nodes of induced subtree modifications in trees derived from NTAs. The *paths_to_roots* part cannot be omitted because the reevaluation starts at the root of the tree and ends as soon as all replaced subtrees are either reevaluated or found in the cache.

Let *VIS* be the mapping from a HAG to visit functions as discussed in section 3. Let T be a tree consistently attributed according to a HAG. Suppose T was attributed by *VIS*(HAG)(T). Let T' be the tree after a subtree modification and suppose T' was attributed by *VIS*(HAG)(T').

Lemma 7.1 *Let* Affected_Visits *be the set of visits that need to be computed and will not be found in the cache when using VIS(HAG)(*T'*) with function caching for visits and hash-consing for trees.*

Then |Affected_Visits| *is* $O(|Affected| + |paths\_to\_roots|)$.

Proof Define the set *Affected_Nodes* to be the set of nodes X in T such that X has an attribute in *Affected*. Clearly, $|Affected\_Nodes| \leq |Affected|$.

Define *Needed_Visits(T')* to be the set of visits needed to evaluate T'. Let *root(v)* denote the root of the subtree that is the first argument of visit function v.

Since the number of visits to a node is bounded by a constant based on the size of the grammar, for all nodes r in T',

$$| \{v \mid v \in Needed\_Visits(T') \wedge root(v) = r\} |$$

is bounded by a constant. The only visits which have to be computed are those that were not computed previously. Therefore,

$$\begin{aligned} Affected\_Visits \subseteq \{v \mid \quad & v \in Needed\_Visits(T') \\ & \wedge root(v) \in (Affected\_Nodes \cup paths\_to\_roots)\} \end{aligned}$$

Therefore,

$$Affected\_Visits \text{ is } O(|Affected\_Nodes| + |paths\_to\_roots|)$$

which is

$$O(|Affected| + |paths\_to\_roots|)$$

□

Theorem 7.1 *Let* Affected_Applications *be the set of function applications that need to be computed and will not be found in the cache when using VIS(HAG)(*T'*) with function caching for visit functions and hash consing for trees. Then,* Affected_Applications *is* $O(|Affected| + |paths\_to\_roots|)$.

Proof Since the number of function calls in a visit is bound by a constant based on the size of the grammar,

$$Affected\_Applications \text{ is } O(|Affected\_Visits|)$$

Using the previous lemma the theorem holds. □

8 Conclusions and future work

A new algorithm for the incremental evaluation of HAGs was presented. Two new approaches are succesfully combined. First, the results of visit functions are cached instead of results of semantic functions. Second, bindings are used containing attribute values computed in earlier visit functions and used by subsequent visit functions visiting the same tree.

Several space-optimizations for bindings are possible. Sharing and efficient caching is achieved by hash consing for trees and bindings.

The resulting algorithm runs in $O(|Affected| + |paths\_to\_roots|)$ steps after subtree modifications, where $|paths\_to\_roots|$ is the sum of the lengths of all paths from the root to all subtree modifications, which is almost as good as an optimal algorithm for first-order AGs (which runs in $O(|Affected|)$).

There are several other ways to improve the performance of which we will mention two here. First, it is possible to split the visit subsequences into independent parts. Then, several independent visit functions for one visit subsequence can be generated. As a consequence only those parts of the attributes will be recomputed of which the input has changed. Second, instead of explicitly representing the tree and calling visit functions to attribute it, the tree is represented *through* one large visit function which is built by pasting together the visit functions of each treenode.

This technique has two major advantages. Firstly, a visit function may be made to depend on precisely that part of the tree that will be visited by it. This increases the number of cache hits. Secondly, copyrules may be removed during the construction phase. This results in shortcircuiting copychains and in minimizing the number of recomputed visit functions.

Acknowledgements

The authors wish to thank Maarten Pennings for carefully reading this document.

References

[Deransart and Jourdan 90] Deransart, P., M. Jourdan (Eds.). *Attribute Grammars and their Applications.* Proceedings of the International Workshop on Attribute Grammars and their Applications (WAGA), LNCS 461, Paris, September 19-21, 1990.

[Deransart, Jourdan and Lorho 88] Deransart, P., M. Jourdan and B. Lorho. *Attribute Grammars: Definitions, Systems and Bibliography.* LNCS 323, Springer Verlag, Aug. 1988.

[Hughes 82] Hughes, R.J.M. *Super-combinators: A New Implementation Method for Applicative Languages.* In Proc. ACM Symp. on Lisp and Functional Progr., Pittsburgh, 1982.

[Kastens 80] Kastens, U. *Ordered Attributed Grammars.* Acta Informatica, 13, pages 229–256, 1980.

[Pugh 88] Pugh, W.W. *Incremental Computation and the Incremental Evaluation of Functional Programs.* Tech. Rep. 88-936 and Ph.D. Thesis, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Aug. 1988.

[Reps and Teitelbaum 88] Reps, T. and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, NY, 1988.

[Swierstra and Vogt 91] Swierstra, S.D. and H.H. Vogt. *Higher Order Attribute Grammars.* In the proceedings of the International Summer School on Attribute Grammars, Applications and Systems, (To Appear), Prague, June 4-13, 1991.

[Teitelbaum and Chapman 90] Teitelbaum, T. and R. Chapman. *Higher-Order Attribute Grammars and Editing Environments.* ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, New York, pages 197-208, June, 1990.

[Turner 1985] Turner, D.A. *Miranda: A non-strict functional language with polymorphic types.* In J. Jouannaud, editor, Funct. Progr. Lang. and Comp. Arch., pages 1-16, Springer, 1985.

[Vogt, Swierstra and Kuiper 89] Vogt, H.H., S.D. Swierstra and M.F. Kuiper. *Higher Order Attribute Grammars.* ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland, Oregon, pages 131-145, June, 1989.

[Vogt, v.d. Berg and Freije 90] Vogt, H.H., A. van den Berg and A. Freije. *Rapid development of a program transformation system with attribute grammars and dynamic transformations.* In the proceedings of the International Workshop on Attribute Grammars and their Applications (WAGA), LNCS 461, Paris, pages 101-115, September 19-21, 1990.

[Yeh 83] Yeh, D. *On incremental evaluation of ordered attributed grammars.* BIT, 23:308-320, 1983.