Higher Order Attribute Grammars

S.D. Swierstra and H.H. Vogt

Technical Report RUU-CS-91-14
September 1991
(revised version)

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

ISSN: 0924-3275

Higher Order Attribute Grammars*

Doaitse Swierstra   Harald Vogt†

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

Abstract

Higher Order Attribute Grammars (HAGs) are an extension of normal attribute grammars in the sense that the distinction between the domain of parse-trees and the domain of attributes has disappeared: parse trees may be computed in attributes and grafted to the parse tree at various places. As a result semantic functions may be described by attribute evaluation.

We will present the basic definitions for HAGs, and compare them with attribute coupled grammars, extended affix grammars and functional programming languages. We will indicate how multi-pass compilers and a compiler for supercombinators can be described this way.

It will be shown that, especially in the case of incremental evaluation, the conventional execution model has to be generalised. Such a model, based on function caching, hash-consing and combinator construction will be discussed. This model encompasses many of the more ad-hoc optimisations one finds in standard implementations of normal attribute grammars.

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, WAGA 90].

*Revised version of the original Lecture Notes (RUU-CS-91-14) of the International Summer School on Attribute Grammars, Applications and Systems (SAGA), Prague, June 4-13, 1991 and a copy of the paper which will appear in the SAGA 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 attribute grammars [Vogt, Swierstra and Kuiper 89] were introduced by promoting abstract syntax trees (i.e. recursive data structures) to "first class citizens":

- they can be the result of a semantic function
- they can be passed as attributes
- they can be grafted into the current tree, and then be attributed themselves, probably resulting in further trees being computed and inserted into the original tree.

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 [TC90]. Our evaluation algorithm [Vogt, Swierstra and Kuiper 91] handles multiple occurrences of the same NTA (and the same subtree) efficiently in $O(|\text{Affected}| + |\text{paths\_to\_roots}|)$ steps, where paths_to_roots is the sum of the lengths of all paths from the root to 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.

The remainder of this article is structured as follows. Section 2 will discuss the shortcomings of conventional AGs. Section 3 defines AGs and (O)HAGs. A compiler for supercombinators is presented in section 4. We discuss related formalisms in section 5. Section 6 presents basic incremental techniques necessary for the incremental evaluator presented in section 7. Finally, section 8 presents a discussion.

2 Shortcomings of AGs

One of the main shortcomings of attribute grammars has been that often a computation has to be specified which is not easily expressable by some form of induction over the abstract syntax tree. The cause for this shortcoming has been the fact that often the grammar used for parsing the input into a data structure dictates the form of the syntax tree. It is however in no way obvious why especially that form of syntax tree would be the optimal starting point for performing further computations.

A further, probably more esthetical than factual, shortcoming of attribute grammars is that there exists usually no correspondence between the grammar part of the system and the (functional) language which is used to describe the semantic functions.

A third shortcome of AGs is that the attributes can be used to diagnose or reject syntax *after* attribuation but cannot be used to guide the syntax *before* attribuation. The following sections will discuss the above mentioned shortcomings in more detail.

2.1 Multi pass/Many pass compilers

The term *compilation* is mostly used to denote the conversion of a program expressed in a human-oriented *source language* into an equivalent program expressed in a hardware-oriented *target language.* A compilation is often implemented as a sequence of transformations *(SL, L_1), (L_1, L_2), ..., (L_k, TL)*, where *SL* is the source language, *TL* the target language and all L_i are called intermediate languages. In attribute grammars *SL* is parsed, a structure tree corresponding with *SL* is build and finally attribute evaluation takes place, the *TL* is obtained as the value of an attribute. So an attribute grammar implements the direct transformation *(SL, TL)* and no special intermediate languages are used. The concept of an intermediate language does not occur naturally in the attribute grammar formalism. Using attributes to emulate intermediate languages is difficult to do and hard to understand. Higher order attribute grammars (HAGs) provide an elegant and powerful solution for this weakness, as attribute values can be used to define the expansion of the structure tree during attribute evaluation.

In a multi-pass compiler compilation takes place in a fixed number of steps, which we will model by computing the intermediate trees as a synthesized attributed of earlier computed trees. These attributes are then used in further attribute evaluation, by grafting them onto the tree on which the attribute evaluator is working. A pictorial description of this process is shown below.

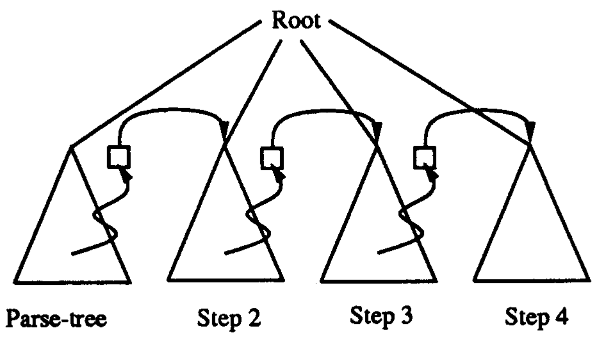


Figure 1: The tree of a 4-pass compiler after evaluation

Attribute Coupled Grammars (ACGs)[Ganzinger and Giegerich 84] exactly define this extension, but nothing more. The Cornell Synthesizer Generator [Reps and Teitelbaum 88] provides only one step: the abstract syntax tree, which is used as the starting point for the attribution is computed as a synthesized attribute of the parse tree. A larger example of the application of this mechanism can be found in [Vogt, Swierstra and Kuiper 89].

2.2 Separate semantic functions

A direct consequence of the dual-formalism approach (attribute grammar part versus semantic functions) is that a lot of properties present in one of the two formalisms are totally absent in the other, resulting in the following anomalities:

- often at the semantic function level considerable computations are being performed which could be more easily expressed by an attribute grammar. It is not uncommon to find descriptions of semantic functions which are several pages long, and which could have been elegantly described by an attribute grammar
- in the case of an incrementally evaluated system the semantic functions do not profit from this incrementality property, and are either being completely evaluated or completely re-used.

Here we will demonstrate the possibility to avoid the use of a separate formalism for describing semantic functions. In Figure 2 a grammar is given 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. Thus the program:

let a,b,c **in** a, c, c, b **ni**

is mapped onto the sequence [1, 3, 3, 2]. In the grammar the inherited and the synthesized attributes of a non-terminal are separated by a $\rightarrow$, and their types have been indicated explicitly. The productions are explicitly labeled, and these labels have been used as semantic functions in constructing the attribute *ENV*. In the example the following can be noted:

- The attribute $\overline{ENV}$ is a higher order attribute. The tree structure is built using the *constructor functions* **env** and **empty_env**, which correspond to the respective productions for *ENV*. The attribute *APPS.env* is instantiated (i.e. a copy of the tree is attributed) in the occurrences of the first production of *APPS*, and takes the rôle of a semantic function.
- The first production rule for the non-terminal *ENV* contains two *initial* attributes. Attributes from this class are initialised by the constructor functions, to which they are passed as an extra set of parameters. This is a syntactic feature which may be considered as an extension of the usual approach in ordinary attribute grammars, where the terminal symbols are allowed to have synthesized attributes which are usually initialised by the scanner.
- Notice that there may exist many instantiations of the *env*-tree, all with different attributes. There thus does not any longer exist an one-to-one correspondence between attributes and abstract syntax trees. As we will see in a later section this has severe consequences for the efficient implementation, which can no longer be a straightforward extension of the conventional evaluators; at the same time however we will see how a number of problems which have to be dealt with separately in these conventional implementations can be handled implicitly by our more general approach.

We finish this section by noticing that any function defined in a functional language can be computed by a HAG without making use of separate semantic functions. This will not be proved here; we will only show as an example a grammar which models the computation of the factorial numbers in Figure 3.

```
$ROOT(\rightarrow \textbf{int}^* \; seq)$
$::=$ block(let $DECLS$ in $APPS$ ni)
        $APPS.env := DECLS.env$
        $ROOT.seq := APPS.seq$

$DECLS(\rightarrow \textbf{int} \; number, \; ENV \; env)$
$::=$ def($DECLS$, $identifier$)
        $DECLS_0.number := DECLS_1.number + 1$
        $DECLS_0.env :=$ env([$identifier.id$, $DECLS_1.number$],$DECLS_1.env$)
|    empty_decls()
        $DECLS.env :=$ empty_env()
        $DECLS.number := 1$

$APPS(ENV \; env \rightarrow \textbf{int}^* \; seq)$
$::=$ use($APPS$, $identifier$, $\overline{ENV}$)
        $APPS_0.seq := APPS_1.seq$ ++[$ENV.index$]
        $\overline{ENV} := APPS_0.env$
        $ENV.param := identifier.id$
        $APPS_1.env := APPS_0.env$
|    empty_use()
        $APPS.seq := [\,]$

$ENV(ID \; param \rightarrow \textbf{int} \; index)$
$::=$ env([$ID \; id$, $\textbf{int} \; number$], $ENV$)
        $ENV_0.index :=$ if $ENV_0.param{=}id \rightarrow number$
                          [] $ENV_0.param{\neq}id \rightarrow ENV_1.index$
                          fi
        $ENV_1.param := ENV_0.param$
|    empty_env()
      $ENV.index :=$ errorvalue
```

Figure 2: A Higher Order Attribute Grammar

```
$ROOT( \rightarrow$ int $res)$
$::=$ start$(integer, \overline{F})$
      $F.n := integer$
      $\overline{F} :=$ loop$()$
      $ROOT.res := F.res$

$F($int $n \rightarrow$ int $res)$
$::=$ loop$(\overline{F})$
      $\overline{F}_1 :=$ if $F_0.n=1 \rightarrow$ stop$()$
            [] $F_0.n \neq 1 \rightarrow$ loop$()$
            fi
      $F_1.n := F_0.n - 1$
      $F_0.res := F_1.res \times F_0.n$
|   stop$()$
      $F.res := 1$
```

Figure 3: Computation of a factorial number by a HAG

2.3 HAGs and editing environments

This section expresses some thoughts about HAGs and editing environments and is almost literally taken from [TC90].

A weakness of the *first-order* attribute-grammar editing model is its strict separation of syntactic and semantic levels, with priority given to syntax. The attributes are completely constrained by their defining equations, whereas the abstract-syntax tree is unconstrained, except by the local restrictions of the underlying context-free grammar. The attributes, which are relied on to communicate context-sensitive information throughout the syntax tree, have no way of generating derivation trees. They can be used to diagnose or reject incorrect syntax *a posteriori* but cannot be used to guide the syntax *a priori.*

A few examples illustrate the desirability of permitting syntax to be guided by attribution:

1. In a forms processing environment, we might want the contents of a male/female field to restrict which other fields appear throughout the rest of a form.

2. In a programming language environment, we might want a partially succesful type inference to provide a declaration template that the user can further refine by manual editing.

3. In a proof development or program transformation environment, we might want a theorem prover to grow the proof tree automatically whenever possible, leaving subgoals for the user to work on wherever necessary.

For more details the reader is referred to [TC90, Vogt, v.d. Berg and Freije 90].

3 Definitions for AGs and (O)HAGs

In this section higher order attribute grammars (HAGs) are being defined. In AGs there exists a strict boundary between attributes and the parse tree. HAGs remove this boundary. A new kind of attributes, so called non-terminal attributes (NTAs), will be defined. These are both non-terminals of the grammar as well as attributes defined by a semantic function. During the initial construction of a parse tree a non-terminal attribute X occurring in the right hand side of a production is considered as a non-terminal for which only the empty production ($X \rightarrow \epsilon$) exists. During attribute evaluation NTA X is assigned a value, which is constrained to be a non-attributed tree derivable from X. As a result of this assignment the original parse tree is expanded with the non-attributed tree computed in the NTA X and its associated attributes are scheduled for computation. A necessary condition for a HAG to be well-formed is that the dependency graph of every possible partial tree does not give rise to circularities; a direct consequence of this is that attributes belonging to an instance of a NTA should not be used in the computation leading to this NTA.

First, attribute evaluation of HAGs is explained, followed by a definition of normal attribute grammars (based on [Waite and Goos 84]) including local attributes. In the next step higher order attribute grammars are defined.

3.1 Attribute evaluation

Evaluation of attribute instances, expansion of the labeled tree (see definition 3.9) and adding new attribute instances is called *attribute evaluation* and might be thought to proceed as follows.

To analyze a string according to its higher order attribute grammar specification, we first construct the labeled tree derived from the root of the higher order attribute grammar. Then evaluate as many attribute instances as possible. As soon as virtual non-terminal instance (see definition 3.8) $\overline{X}$ is computed, expand the labeled tree derived from the root at the corresponding leaf $\overline{X}$ with the labeled tree in $\overline{X}$ and add the attribute instances resulting from the expansion. The virtual non-terminal $\overline{X}$ has now become an instantiated non-terminal $\overline{X}$. Continue the evaluation until there are no more attribute instances to evaluate and all possible expansions have been performed.

The order in which attributes are evaluated is left unspecified here, but is subjected to the constraint that each semantic function is evaluated only when all its argument attributes have become available. When all the arguments of an unavailable attribute instance have become available, we say it is *ready for evaluation.*

Using the definition of attribute evaluation and the observation to maintain a work-list S of all attribute instances that are ready for evaluation we get, as is stated in [Knuth 68, Knuth 71] and [Reps 82], the following Attribute evaluation algorithm of Figure 4.

The difference with the algorithm defined by [Reps 82] is that the labeled tree T can be expanded during semantic analysis. This means that if we evaluate a NTA $\overline{X}$, we have to expand the tree at the corresponding leaf $\overline{X}$ with the tree computed in $\overline{X}$. Furthermore,

```
procedure evaluate(T: an unevaluated labeled tree)
let  D = a dependency relation on attribute instances
     S = a set of attribute instances that are ready for evaluation
     α, β = attribute instances
in
D := DT(T) { the dependency relation over the tree T }
S := the attribute instances in D which are ready for evaluation
while S ≠ ∅ do
     select and remove an attribute instance α from S
     evaluate α
     if α is a NTA of the form $\overline{X}$
     then expand T at $\overline{X}$ with the unevaluated tree in α
          D := D ∪ DT(X)
          S := S ∪ the attribute instances in DT(X) ready for evaluation
     fi
     forall β ∈ successor(α) in D do
          if β is ready for evaluation
          then insert β in S
          fi
     od
od
```

Figure 4: Attribute evaluation algorithm

the new attribute instances and their dependencies of the expansion (the set $DT(X)$) have to be added to the already existing attribute instances and their dependencies, and the work-list S must be expanded by all the attribute instances in $DT(X)$ that are ready for evaluation.

3.2 Definition of AGs

A context free grammar $G = (T, N, P, Z)$ consists of a set of terminal symbols T, a set of non-terminal symbols N, a set of productions P and a start symbol $Z \in N$. To every node in a structure tree corresponds a production from G.

Definition 3.1 *An* attribute grammar *is a 3-tuple* $AG = (G, A, R)$. $G = (T, N, P, Z)$ *is a context free grammar.*
$A = \bigcup_{X \in T \cup N} AIS(X) \cup \bigcup_{p \in P} AL(p)$ *is a finite set of attributes,*
$R = \bigcup_{p \in P} R(p)$ *is a finite set of attribution rules.*
$AIS(X) \cap AIS(Y) \neq \emptyset$ *implies* $X = Y$. *For each occurrence of non-terminal* X *in the structure tree corresponding to a sentence of* $L(G)$*, exactly one attribution rule is applicable for the computation of each attribute* $a \in A$.

Elements of $R(p)$ have the form

$$\alpha := f(\ldots,\gamma,\ldots).$$

In this attribution rule, f is the name of a function, α and γ are attributes of the form $X.a$ or $p.b$. In the latter case $p.b \in AL(p)$. In the sequel we will use the notation b for $p.b$ whenever possible. We assume that the functions used in the attribution rules are strict in all arguments.

Definition 3.2 *For each $p : X_0 \rightarrow X_1 \ldots X_n \in P$ the set of* defining occurrences *of attributes is*

$$AF(p) = \{X_i.a \mid X_i.a := f(\ldots) \in R(p)\}$$
$$\cup \{p.b \mid p.b := f(\ldots) \in R(p)\}$$

An attribute $X.a$ is called synthesized *if there exists a production $p : X \rightarrow \chi$ and $X.a$ is in $AF(p)$; it is* inherited *if there exists a production $q : Y \rightarrow \mu X \nu$ and $X.a \in AF(q)$. An attribute b is called* local *if there exists a production p such that $p.b \in AF(p)$.*

AS(X) is the set of synthesized attributes of X. *AI(X)* is the set of inherited attributes of X. *AL(p)* is the set of local attributes of production p.

Definition 3.3 *An attribute grammar is* complete *if the following statements hold for all X in the vocabulary of G:*

- *For all $p : X \rightarrow \chi \in P, AS(X) \subseteq AF(p)$*
- *For all $q : Y \rightarrow \mu X \nu \in P, AI(X) \subseteq AF(q)$*
- *For all $p \in P, AL(p) \subseteq AF(p)$*
- $AS(X) \cup AI(X) = AIS(X)$
- $AS(X) \cap AI(X) = \emptyset$

Further, if Z is the root of the grammar then AI(Z) *is empty.*

Definition 3.4 *An attribute grammar is* well defined *(WAG) if, for each structure tree all attributes are effectively computable.*

Definition 3.5 *For each $p : X_0 \rightarrow X_1 \ldots X_n \in P$ the set of* strict attribute dependencies *is given by*

$$DDP(p) = \{(\beta,\alpha) \mid \alpha := f(\ldots\beta\ldots) \in R(p)\}$$

where α and β are of the form $X_i.a$ or b. The grammar is locally acyclic *if the graph of DDP(p) is acyclic for each $p \in P$.*

We often write $(\alpha,\beta) \in DDP(p)$ as $(\alpha \rightarrow \beta) \in DDP(p)$, and follow the same conventions for the relations defined below. If no misunderstanding can occur, we omit the specification of the relation. We obtain the complete dependency graph for a labeled structure tree by "pasting together" the direct dependencies according to the syntactic structure of the tree.

Definition 3.6 *Let S be the attributed structure tree, and let $K_0 \ldots K_n$ be the nodes corresponding to an application of $p : X_0 \to X_1 \ldots X_n$ and γ, δ attributes of the form $K_i.a$ or b corresponding with the attributes α, β of the form $X_i.a$ or b. We write $(\gamma \to \delta)$ if $(\alpha \to \beta) \in DDP(p)$. The set $DT(S) = \{(\gamma \to \delta)\}$, where we consider all applications of productions in S, is called the* dependency relation over the tree *S.*

The following theorem gives another characterization of well-defined attribute grammars. A proof can be found in [Waite and Goos 84].

Theorem 3.1 *An attribute grammar is* well-defined *iff it is complete and the graph $DT(S)$ is a-cyclic for each structure tree S.*

3.3 Definition of Higher order AGs

An higher order attribute grammar is an attribute grammar with the following extensions:

Definition 3.7 *For each $p : X_0 \to X_1 \ldots X_n \in P$ the set of* non-terminal attributes (NTAs) *is defined by*

$$NTA(p) = \{X_j \mid X_j := f(\ldots) \in R(p)\}$$

Because a non-terminal attribute is also an attribute, an actual tree may contain NTAs (not yet computed non-terminal attributes) as leafs. Therefore we change the notion of a tree. Two kinds of non-terminals are distinguished, virtual non-terminals (NTAs without a value) and instantiated non-terminals (NTAs with a value and normal non-terminals).

Definition 3.8 *A non-terminal instance X in a tree is called*

- *a* virtual *non-terminal if $X \in \bigcup_{p \in P} NTA(p)$ and the function defining X has not yet been evaluated*
- *an* instantiated *non-terminal if $X \notin \bigcup_{p \in P} NTA(p)$ or $X \in \bigcup_{p \in P} NTA(p)$ and the function defining X has been evaluated*

Definition 3.9 *A* labeled *tree is defined as follows*

- *the leafs of a labeled tree are labeled with terminal or virtual non-terminal symbols*
- *the nodes of a labeled tree are labeled with instantiated non-terminal symbols*

From now on, the terms "structure tree" and "labeled structure tree" are all used to refer to a labeled tree. In the text a non-terminal attribute X will be indicated as $\overline{X}$.

Definition 3.10 *A semantic function f in a rule $\overline{X} := f(\ldots)$ is* correctly typed *if f returns a term representing a parse tree derivable from X*

This definition will be used to ensure that a NTA $\overline{X}$ will be expanded with a labeled tree which is derivable from X. Note that a check whether a function is correctly typed can be done statically.

Definition 3.11 *An higher order attribute grammar is* complete *if the underlying AG is complete and the following holds for all productions $p : Y \rightarrow \mu \in P$:*

- *$NTA(p) \subseteq AL(p)$*

and for all $\overline{X} \in NTA(p)$:

- $\overline{X} \in \mu$
- *For all rules $\alpha := f(\gamma)$ in $R(p)$, $\overline{X} \notin \gamma$*
- *For all rules $\overline{X} := f(\gamma)$ in $R(p)$, f is correctly typed.*

The above definition defines NTAs as local attributes which only occur as a non-terminal at the right-hand-side of a production and as an attribute at the left-hand-side of a semantic function. If we look at the Attribute evaluation algorithm in Figure 4, there are two potential problems:

- non-termination
- attribute instances may not receive a value

The algorithm might not terminate if the labeled tree grows indefinitely, in which case there will always be virtual non-terminal attribute instances which can be instantiated (Figure 6). There are two reasons why an attribute might not receive a value:

- a cycle shows up in the dependency relation D: attribute instances involved in the cycle will never be ready for evaluation, so they will never receive a value.
- there is a non-terminal attribute instance, say $\overline{X}$, which depends on a synthesized attribute of $\overline{X}$.

The second reason may deserve some explanation. Suppose we have a tree T containing rule p and $\overline{X}$ is a non-terminal attribute instance in T. Furthermore the dependency relation D of all the attribute instances in T contains no cycles (Figure 5).

If we take a closer look at node $\overline{X}$ in T, then if $\overline{X}$ doesn't depend on synthesized attributes of $\overline{X}$ it can be computed. But should $\overline{X}$ depend on synthesized attributes of $\overline{X}$, as in Figure 5 it can't be computed. This is because the synthesized attributes of $\overline{X}$ are computed after the tree is expanded. So a non-terminal attribute should nor directly nor indirectly depend on its own synthesized attributes. To prevent this we let every synthesized attribute of $\overline{X}$ depend on $\overline{X}$. Therefore the set of extended direct attribute dependencies is defined.

Definition 3.12 *For each $p : X_0 \to X_1 \dots X_n \in P$ the set of* extended direct attribute dependencies *is given by*

$$EDDP(p) = \{(\alpha \to \beta) \mid \beta := f(\dots\alpha\dots) \in R(p)\}$$
$$\cup \; \{(\overline{X} \to \gamma) \mid \overline{X} \in NTA(p) \text{ and } \gamma \in AS(X)\}$$

Thus a non-terminal attribute can be computed if and only if the dependency relation D (using the *EDDPs*) contains no cycles. This result is stated in the following lemma.

Lemma 3.1 *Every virtual non-terminal attribute will be computed if and only if there will be no cycles in D (using the EDDP) during attribute evaluation.*

Proof The use of *EDDP(p)* prohibits a non-terminal attribute β to be defined in terms of attribute instances in the tree which will be computed in β. Suppose β, which is of the form X, depends on attributes in the tree which is constructed in β. The only way to achieve this is that β somehow depends on the synthesized attributes of X, but by definition of *EDDP(p)* all the synthesized attributes of X depend on β and we have a cycle.

□

```
$R(\to)$
$::=$ p$(\overline{X})$
        $\overline{X} := f(X.s)$
$X(\to$ **int** $s)$
$::=$ q$(one)$
        $X.s := 1$
|    r$(two)$
        $X.s := 2$
```

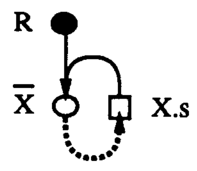


Figure 5: The non-terminal attribute can't be computed, a cycle occurs if the extra dependency is added (dashed arrow)

Definition 3.13 *An higher order attribute grammar is* well-defined *if, for each labeled structure tree S all attributes are effectively computable using the algorithm in Figure 4.*

It is clear that if D never contains a cycle during attribute evaluation, all the (non-terminal) attribute instances are effectively computable. Whether they will eventually be computed depends on the scheduling algorithm used in selecting elements from the set S. It is generally undecidable whether a given HAG will have only finite expansions. For instance whether the grammar in Figure 6 has only finite expansions is undecidable. The

```
R( → )
::= r(X)
X( → )
::= loopA(Ā)
      Ā := if f(...) → loopX()
           []        → stop()
           fi
A( → )
::= loopX(X̄)
      X̄ := loopA()
|   stop()
```

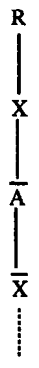


Figure 6: Finite expansion is not guaranteed

tree may grow indefinitely depending on the function f. For a more detailed discussion about well-defined-ness the reader is referred to [Vogt, Swierstra and Kuiper 89].

We used the terms "attribute evaluation" and "attribute evaluation algorithm" to define whether an AG is well-defined. Instead of using an algorithm we could have defined a relation on labeled trees, indicating whether a non-attributed labeled tree is well-defined. We used the algorithm because from that it is easy to derive conditions by which it can be checked whether a HAG is well-defined.

3.4 Ordered HAGs (OHAGs)

In [Kastens 80] a condition is described for well-defined attribute grammars (WAGs): The semantic rules of an AG are well-defined if and only if there is no sentence in the language with circularly dependent attributes. In [Jazayeri 1975] it was proved that deciding whether an AG is well-defined is an exponential problem. In [Kastens 80] ordered attribute grammars (OAGs) were defined: an attributed grammar is ordered if for each symbol a total order over the associated attributes can be found, such that in any context of the symbol the attributes may be evaluated in that order. A specific algorithm is given to construct a total order out of a partial order which describes the possible dependencies between the attributes of a non-terminal. If the thus found total order does not introduce circularities the grammar is called *ordered.* This property can be checked by an algorithm, which depends polynomially in time on the size of the grammar.

Visitsequences are computed from these total orders and the *DPP(p)*'s. These visitsequences define for each productions a total order on the defining attributes occurrences in that production, which determines the order in which these attributes may be computed.

An ordered HAG is now characterized by the following condition: a similar total order on the defining attribute occurrences in a production p can be defined. It determines a fixed sequence of computation for the defining attribute occurrences, applicable in any

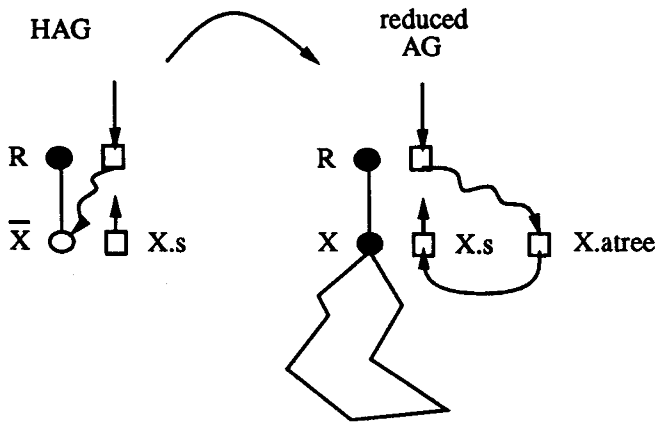


Figure 7: The same part of a structure tree in a HAG and the corresponding reduced AG

tree node labelled with production p.

In this subsection a condition, based on OAGs, is given which may be used to check whether a HAG is ordered.

3.5 Deriving partial orders from AGs

To decide whether a HAG is ordered the HAG is transformed into an AG and it is checked whether the AG is an OAG. The derived orders on defining attribute occurrences in the OAG can be easily transformed to orders on the defining occurrences of the HAG.

In the previous section (Lemma 3.1) it was shown that the *EDDP* ensured that every NTA could be computed. The reduced AG of a HAG is now defined as follows:

Definition 3.14 *Let H be a HAG. The* reduced AG *H' is the result of the following transformations to H:*

1. *in all right hand sides of the productions all occurrences of $\overline{X}$ are replaced by the corresponding X*
2. *all thus converted non-terminals are equipped with an extra inherited attribute* X.atree
3. *all occurrences $\overline{X}$ in the left hand side of the attribution rules are replaced by* X.atree
4. *all synthesized attributes of previously NTAs $\overline{X}$ now contain the attribute X.atree in the right-hand-side of their defining semantic function and are thus explicitly made depending on this attribute.*

The transformation is demonstrated in Figure 7. This definition ensures that all synthesized attributes of NTA $\overline{X}$ (X.atree in the reduced AG) in the HAG can be only computed after NTA $\overline{X}$ (X.atree in the reduced AG) is computed.

Theorem 3.2 *A HAG is ordered if the corresponding reduced AG is an OAG.*

Proof Map the occurrences of X.atree in the orders of the reduced AG derived from a HAG to NTAs $\overline{X}$. The result are orders for the HAG in the sense that the HAG is ordered.

□

We note that this procedure may result in a HAG being rejected, because the derived AG is not ordered; the test may be too pessimistic. Sometimes a HAG is ordered although the reduced AG is not an OAG, as is shown in Figure 8.

$R(\rightarrow)$
$::= \mathsf{p}(\overline{AA})$
$\quad \overline{A}[0] := R.a\_tree$
$\quad \overline{A}[1] := R.a\_tree$
$A(\rightarrow)$
$::= \mathsf{q}(zero)$
$\mid \quad \mathsf{r}(one)$

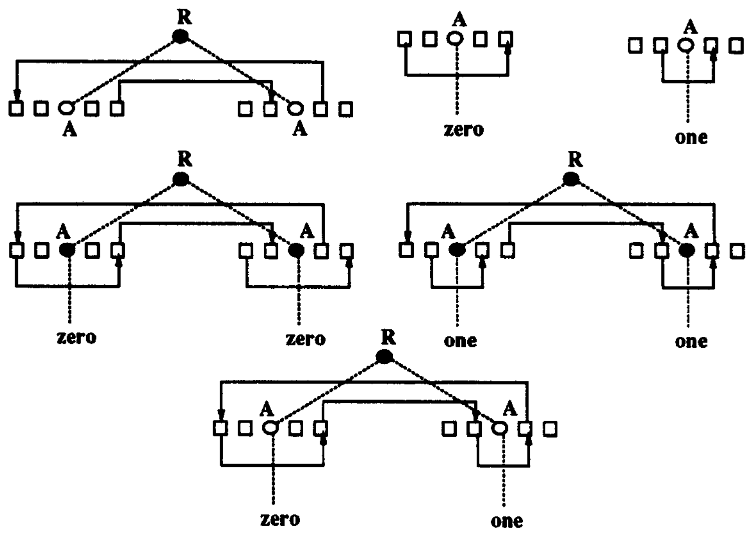


Figure 8: The lowest tree shows a cycle in the attribute dependencies which is only possible in the reduced AG

The class of OAGs is a sufficiently large class for defining programming languages, and it is expected that the above described way to derive evaluation orders for OHAGs provides a large enough class of HAGs.

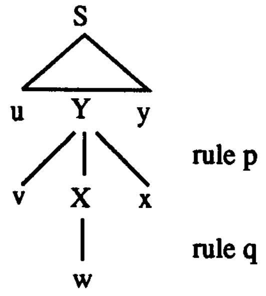
3.6 Visitsequences for an OHAG

The difference with visitsequences as they are defined by [Kastens 80] for an OAG is that in a HAG the instruction set is extended with an instruction to evaluate a non-terminal attribute and expand the labeled tree at the corresponding virtual non-terminal. The following introduction to visitsequences for a HAG is almost literally taken from [Kastens 80].

The evaluation order is the base for the construction of a flexible and efficient attribute evaluation algorithm. 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

$$S \Rightarrow uYy \rightarrow_p uvXxy \rightarrow_q uvwxy \Rightarrow s.$$

Then the corresponding part of the structure tree is



An attribute evaluation algorithm traverses the structure tree using the operations "move down to a descendant node" (e.g. from K_Y to K_X) or "move up to the ancestor node" (e.g. from K_X to K_Y). During a visit of node K_Y some attributes of $AF(p)$ are evaluated according to semantic functions, if p is applied at K_Y. In general several visits to each node are needed until all attributes are evaluated. A local tree walk rule is associated to each p. It is a sequence of four types of moves: 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 certain non-terminal attribute. The last instruction is specific for a HAG.

Visitsequences for a HAG can be easily derived from visitsequences of the corresponding reduced AG. In an OAG the visitsequences are derived from the evaluation order on the defining attribute occurrences. A description of the computation of the visitsequences in an OAG is given in [Kastens 80]. The visitsequence of a production p in an AG will be denoted as *VS(p)* and in the HAG as *HVS(p)*.

Definition 3.15 *Each* visitsequence *VS(p) associated to a rule $p \in P$ in an AG is a linearly ordered relation over defining attribute occurrences and visits.*

$$VS(p) \subseteq AV(p) \times AV(p),\ AV(p) = AF(p) \cup V(p)$$

$$V(p) = \{v_{k,i} | 0 \leq i \leq \|p\|, 1 \leq k \leq nov_X, X = X_i\}$$

$v_{k,0}$ denotes the k-th ancestor visit, $v_{k,i}$, $i > 0$ denotes the k-th visit of the descendant X_i, $\|p\|$ denotes the number of non-terminals in production p and nov_X denotes the number of visits that will be made to X. For the definition of *VS(p)* see [Kastens 80]. We now define the *HVS(p)* in terms of the *VS(p)*.

Definition 3.16 *Each* visitsequence *HVS(p) associated to a rule $p \in P$ in a HAG is a linearly ordered relation over defining attribute occurrences, visits and expansions.*

$$HVS(p) \subseteq HAV(p) \times HAV(p),\ HAV(p) = AV(p) \cup VE(p)$$

$$VE(p) = \{e_i \mid 1 \leq i \leq \|p\|\}$$

where AV(p) is defined as in the previous definition.

$$HVS(p) = \{g(\gamma) \rightarrow g(\delta) \mid (\gamma \rightarrow \delta) \in VS(p)\}$$

with $g : AV(p) \rightarrow HAV(p)$ defined as

$$g(a) = \begin{cases} e_i & \textit{if a is of the form } X_i.atree \\ a & \textit{otherwise} \end{cases}$$

e_i denotes the computation of the non-terminal attribute X_i and the expansion of the labeled tree at $\overline{X}_i$ with the tree computed in X_i.

Note that a descendant of a virtual non-terminal can only be visited *after* the virtual non-terminal is instantiated. The visitsequences for OAGs are defined in such a way that during a visit to a node one or more synthesized attributes are computed. Because all synthesized attributes of a virtual non-terminal $\overline{X}$ depend by construction on the non-terminal attribute, the corresponding attribute *X.atree* in the OAG will be computed before the first visit.

In [Kastens 80] it is proved that the check and the computation of the visitsequences *VS(p)* for an OAG depends polynomially in time on the size of the grammar. The mapping from the HAG to the reduced AG and the computation of the visitsequences *HVS(p)* depend also polynomially in time on the size of the grammar. So the subclass of well-defined HAGs derived by computation of the reduced AG, analyzing whether the reduced AG is an OAG and computation of the visitsequences for an HAG can be checked in polynomial time. Furthermore an efficient and easy to implement algorithm, as for OAGs, based on visitsequences can be used to evaluate the attributes in a HAG.

4 A compiler for supercombinators[1]

In this example we will give a description of the translation of a λ-expression into super combinator form. The purpose of this section is two-fold. In the first place it serves as an example of the use of higher order attribute grammars. In the second place it will serve as an introduction to the use of combinators, an implementation technique on which the later to be discussed incremental evaluation is based.

In implementing the λ-calculus, one of the basic mechanisms which has to be provided for is the β-reduction, informally defined as a substitution of the parameter in the body of a function by the argument expression.

In the formal semantics of the calculus this substitution is defined as a string replacement. It will be obvious that implementing this string replacement as such is undesirable and inefficient. We easily recognise the following disadvantages:

1. the basic steps of the interpreter are not of more or less equal granularity
2. the resulting string may contain many common subexpressions which, when evaluated, all result in the same value

[1]This section is completely based on *Generating Supercombinator Code using Higher Order Attribute Grammars*, Maarten Pennings and Ben Juurlink, Department of Computer Science, Utrecht University.

3. large parts of the body may be copied and submitted to the substitution process, which are not further reduced in the future but instead are being discarded because of the rewriting of an **if-then-else-fi** reduction rule

4. because substitutions may define the value of global variables of λ-expressions defined in the body of a function, the value of these bodies may change during the evaluation process. It is thus almost impossible to generate code which will perform the copying and substitution for this inner λ-expression.

The second of these disadvantages may be solved by employing graph-reduction instead of string reduction. Common sub-expressions may be shared in this representation. To remedy the other three problems [Turner 79a] shows how any lambda-expression may be compiled into an equivalent expression consisting of **SKI**-combinators and standard functions only. In the resulting implementation the expressions are copied and substituted "by need" by applying the simple reduction rules associated with these combinators. Although the resulting implementation, using graph reduction, is very elegant, it leads to an explosion in the number of combinator occurrences and thus of basic reduction steps. In [Hughes 85] supercombinators are introduced; although the first and third problem are not solved its advantages in solving the fourth problem are such that it is still considered an attractive approach.

In this section we will describe a compiler for lambda-expressions to supercombinator code in terms of higher order attribute grammars. The algorithm is based on [Hughes 82].

The basic idea of a super-combinator is to define for each function which refers to global variables, an equivalent function to which the global variables are being passed explicitly. The resulting function is called a combinator, because it does not contain any free variables any more. At the reduction all the global variables and the actual argument are substituted in a single step. Because the code of the function may be considered as an invariant of the reduction process it is possible to generate machine code for it, which takes care of construction of the graph and the substitution process.

The situation has then become fairly similar to the conventional stack implementations of procedural languages, where the entire context is being passed (usually called the *static link*) and the appropriate global values are being selected from that context by indexing instructions. The main difference is that not the entire environment is being passed, but only those parts which are explicitly being used in the body of the function. As a further optimisation subexpressions of the body, which do not depend on the parameter of the function, are abstracted and passed as an extra argument. As a consequence their evaluation may be shared between several invocations of the same function.

4.1 Lambda expressions

As an example consider the lambda expression $f = [\lambda x : [\lambda y : \oplus \cdot ([\lambda z : z \cdot (x \cdot y \cdot y) \cdot (z \cdot (\sigma \cdot y) \cdot y)] \cdot x) \cdot 7]]$. In this expression $\oplus$, σ and 7 are constant functions, e.g. the add and successor operation, and the number 7. Note that

$$f \cdot \otimes \cdot a \quad = \quad \oplus \cdot ([\lambda z : z \cdot (\otimes \cdot a \cdot a) \cdot (z \cdot (\sigma \cdot a) \cdot a)] \cdot \otimes) \cdot 7$$

$$= \oplus \cdot (\otimes \cdot (\otimes \cdot a \cdot a) \cdot (\otimes \cdot (\sigma \cdot a) \cdot a)) \cdot 7$$

Expression f may be thought of as a tree. This mapping is a one to one since we assume application $(\cdot)$ to be left-associative. The corresponding abstract syntax tree—in linear notation—has the form

```
lop(x,lop(y,lap(lap(lco(⊕),lap(lop(z,lap(lap(lid(z),lap(lap(lid(x),lid(y)),lid(y)))
                                      ,lap(lap(lid(z),lap(lco(σ),lid(y))),lid(y))
                                )  )
                           ,lid(x)
                 )         )
              ,lco(7)
    )   )   )
```

where we use the following definition for type *lexp* representing lambda-expressions

lexp ::=	lop(*id*, *lexp*)	{λ-introduction}
|	lap(*lexp*, *lexp*)	{function application}
|	lid(*id*)	{identifier occurrence}
|	lco(*id*)	{constant occurrence}

The type *id* is a standard type, representing identifiers. Another standard type is *num*; it is used to represent natural numbers. In order to model the binding process we will introduce a mapping from trees labeled with identifiers (*id*) to trees labeled with naturals (*num*) instead:

```
name == num
nexp ::= nop(name, nexp)
      |  nap(nexp, nexp)
      |  nid(name)
      |  nco(id)
```

In this conversion, identifiers are replaced by a number indicating the "nesting depth" of the bound variable. Hence, x, y, and z from our example will be substituted by 1, 2, and 3 respectively. Constants are simply copied. Although this mapping could be formulated in any "modern" functional language, we are striving for a higher order attribute grammar, so this is a good point to start from.

The non-terminal *lexp* will have two attributes. The first, an inherited one, will contain the *environment*, i.e. the bound variables found so far associated with their nesting level. A list of *id*'s with index-determination ($l^{-1}(i)$) suits our needs (note that $[x, y, z]^{-1}(x) = 1$). The second attribute, a synthesized one, returns the "number-tree" of the above given type *nexp*.

```
$env == [id]$
$lexp<\downarrow env, \uparrow nexp>$
$lexp<\downarrow e_0, \uparrow n_0>$ ::= $\mathsf{lop}(id, lexp<\downarrow e_1, \uparrow n_1>)$
                          $\neg(id \text{ in } e_0);$
                          $e_1 := e_0{+}{+}[id];$
                          $n_0 := \mathsf{nop}(e_1^{-1}(id), n_1)$
                       |  $\mathsf{lap}(lexp<\downarrow e_1, \uparrow n_1>, lexp<\downarrow e_2, \uparrow n_2>)$
                          $e_1 := e_0;\ e_2 := e_0;$
                          $n_0 := \mathsf{nap}(n_1, n_2)$
                       |  $\mathsf{lid}(id)$
                          $id \text{ in } e_0;$
                          $n_0 := \mathsf{nid}(e_0^{-1}(id))$
                       |  $\mathsf{lco}(id)$
                          $n_0 := \mathsf{nco}(id)$
```

Since we will follow the convention that the startsymbol of a (higher order) attribute grammar cannot have inherited attributes we introduce an extra non-terminal *start*:

```
$start<\uparrow nexp>$
$start<\uparrow n_0>$ ::= $\mathsf{root}(lexp<\downarrow e, \uparrow n_1>)$
                  $e := [\,];$
                  $n_0 := n_1$
```

The lambda expression we gave at the start of this paragraph "returns" the following attribute:

```
nop(1,nop(2,nap(nap(nco(⊕),nap(nop(3,nap(nap(nid(3),nap(nap(nid(1),nid(2)),nid(2)))
                                         ,nap(nap(nid(3),nap(nco(σ),nid(2))),nid(2))
                                    )    )
                               ,nid(1)
                   )           )
               ,nco(7)
   )   )   )
```

4.2 Supercombinators

Before starting to generate super-combinator code we would like to stress that it is easier to derive supercombinator code from *nexp* shaped expressions than from *lexp* shaped expressions. Thus, the supercombinator code generator attributes the *nexp*-tree, not the *lexp*-tree. This is were higher order attribute grammars come into use for the first time: the generated *nexp* tree is substituted for a non-terminal attribute.

$$
\begin{array}{l}
\mathit{start}{<}{\uparrow}\mathit{cexp}{>} \\
\mathit{start}{<}{\uparrow}c_0{>} ::= \mathsf{root}(\mathit{lexp}{<}{\downarrow}e,{\uparrow}n{>}, \overline{\mathit{nexp}}{<}{\uparrow}c_1{>}) \\
\qquad\qquad\qquad e := [\,]; \\
\qquad\qquad\qquad \overline{\mathit{nexp}} := n; \\
\qquad\qquad\qquad c_0 := c_1
\end{array}
$$

The non-terminal *nexp* has a synthesized attribute of type *cexp*. This type, representing supercombinator code, is defined as

$$
\begin{array}{l}
\mathit{params} == [\mathit{name}] \\
\mathit{cexp} ::= \mathsf{cop}(\mathit{params}, \mathit{cexp}) \\
\qquad\quad |\ \ \mathsf{cap}(\mathit{cexp}, \mathit{cexp}) \\
\qquad\quad |\ \ \mathsf{cid}(\mathit{name}) \\
\qquad\quad |\ \ \mathsf{cco}(\mathit{id})
\end{array}
$$

As may be seen from the above definition, combinators generally have multiple parameters. With $\mathsf{cop}([3,1,2], E)$ we denote a combinator with three dummies. In standard notation this would be written as $[\Lambda 312 : E]$ which is equivalent to $[\lambda 3 : [\lambda 1 : [\lambda 2 : E]]]$.

Let us have a closer look at expression $e = [\lambda z : z \cdot (x \cdot y \cdot y) \cdot (z \cdot (\sigma \cdot y) \cdot y)]$ which is a subexpression of our previous example. Any subexpression of (the body of) e that does not contain the bound variable (z) is called *free*. So x, y, σ, $x \cdot y$, $\sigma \cdot y$, and $x \cdot y \cdot y$ are free expressions. Such expressions can be abstracted out, an example being $f = [\Lambda 1234 : 4 \cdot (1 \cdot 2) \cdot (4 \cdot 3 \cdot 2)] \cdot (x \cdot y) \cdot y \cdot (\sigma \cdot y)$.

This transformation from e to f improves the program since, for example, $x \cdot y$ only needs to be evaluated once, rather than every time f is called. Of course f is not optimal yet: the best result emerges when all *maximal* free expressions are abstracted out.

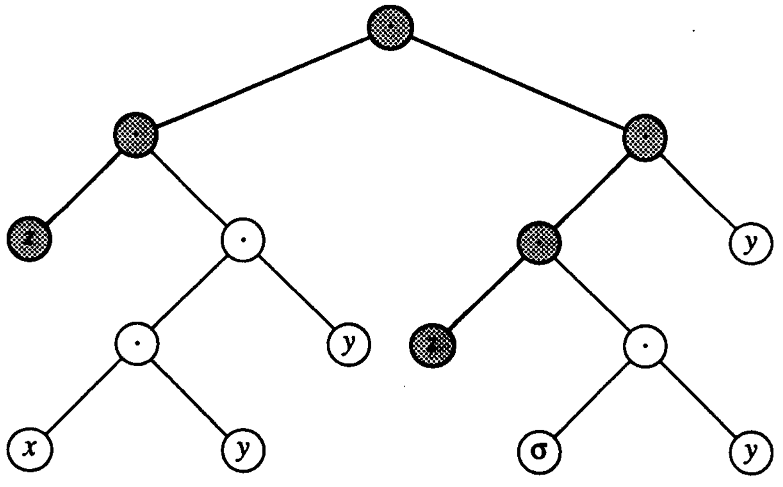


Figure 9: The paths from the root to the tips containing the current dummy are indicated by thick lines thus clearly isolating the maximal free expressions.

As may be seen from Figure 9, $x \cdot y \cdot y$, $\sigma \cdot y$, and y are maximal free expressions. In order to generate the supercombinator for e, each maximal free expression is replaced by some dummy. We reserve the index "0" for the actual parameter introduced by the λ.

$$[\Lambda z : z \cdot \underbrace{(x \cdot y \cdot y)}_{1} \cdot (z \cdot \underbrace{(\sigma \cdot y)}_{2} \cdot \underbrace{y}_{3})]$$

Hence we find as a possible supercombinator:

$$\alpha = [\Lambda 1230 : 0 \cdot 1 \cdot (0 \cdot 2 \cdot 3)]$$

with bindings $\{1 \mapsto x \cdot y \cdot y \ , \ 2 \mapsto \sigma \cdot y \ , \ 3 \mapsto y\}$ so that e equals

$$\alpha \cdot (x \cdot y \cdot y) \cdot (\sigma \cdot y) \cdot y$$

We will now describe an algorithm which finds all maximal free expressions. We could associate a boolean with each expression indicating the presence of the current parameter in the expression. This attribution then depends on this parameter. So, if we are interested in the maximal free expressions of the surrounding expression, we would have to recalculate these attributes.

We use another approach instead: a level is associated with each expression indicating the nesting depth of the most local variable occurring in that expression. If this depth equals the nesting depth of the current parameter, the expression contains this parameter as a subexpression and hence it is not free. Since we substituted all identifiers in *lexp* by a unique number indicating their depth, the level of an expression simply is the maximum of all numbers occurring in that expression.

```
$level == num$
$cexp<\uparrow level>$
$cexp<\uparrow l_0>$ ::= cop($params$, $cexp<\uparrow l_1>$)
                 $l_0 := 0$
               | cap($cexp<\uparrow l_1>$, $cexp<\uparrow l_2>$)
                 $l_0 := l_1 \max l_2$
               | cid($name$)
                 $l_0 := name$
               | cco($id$)
                 $l_0 := 0$
```

Combinators and constants form a special group. They contain no free variables so their level is set to 0, the "most global level"—the unit element of "max". On the other hand, there is no need to abstract out expressions of level 0, since they are irreducable. They form the basis of the functional programming environment.

As a next step, let us concentrate on generating the bindings. A binding is a pair $n \mapsto c$ with $n \in name$ and $c \in cexp$. Since no variable may be bound more than once, we need to know which variables are already bound when we need a new binding. So, we introduce an "environment-in" (initially empty) and an "environment-out" (returning all maximal free subexpressions).

```
$bind == \{name \mapsto cexp\}$
$cexp{<}\uparrow level, \downarrow name, \downarrow bind, \uparrow bind, \uparrow cexp{>}$
$cexp_0{<}\uparrow l_0, \downarrow n_0, \downarrow i_0, \uparrow o_0, \uparrow c_0{>}$
                 $::=$ $\mathsf{cop}(params, cexp_1)$
                       $l_0 := 0;$
                       $o_0 := i_0;\ c_0 := cexp_0$
                 |     $\mathsf{cap}(cexp_1{<}\uparrow l_1, \downarrow n_1, \downarrow i_1, \uparrow o_1, \uparrow c_1{>}, cexp_2{<}\uparrow l_2, \downarrow n_2, \downarrow i_2, \uparrow o_2, \uparrow c_2{>})$
                       $n_1 := n_0;\ n_2 := n_0;$
                       $l_0 := l_1 \max l_2;$
                       **if** $(l_0 = n_0) \vee (l_0 = 0)$
                          **then** $i_1 := i_0;\ i_2 := o_1;\ o_0 := o_2;\ c_0 := \mathsf{cap}(c_1, c_2)$
                          **else** $o_0 := i_0 \sqcup \{|i_0| + 1 \mapsto cexp_0\};\ c_0 := \mathsf{cid}(o_0^{-1}(cexp_0))$
                       **fi**
                 |     $\mathsf{cid}(name)$
                       $l_0 := name;\ \{\ l_0 > 0\ \}$
                       **if** $l_0 = n_0$
                          **then** $o_0 := i_0;\ c_0 := \mathsf{cid}(0)$
                          **else** $o_0 := i_0 \sqcup \{|i_0| + 1 \mapsto cexp_0\}; c_0 := \mathsf{cid}(o_0^{-1}(cexp_0))$
                       **fi**
                 |     $\mathsf{cco}(name)$
                       $l_0 := 0;$
                       $o_0 := i_0;\ c_0 := cexp_0$
```

Since we are not interested in the body of a combinator, we leave out the attributes of $cexp_1$ in $\mathsf{cop}(params, cexp_1)$. The operator $\sqcup$ is defined as follows:

$$S \sqcup \{n \mapsto c\} \ := \textbf{if } c \in \text{rng}(S) \textbf{ then } S \textbf{ else } S \cup \{n \mapsto c\} \textbf{ fi}$$

thus performing common-subexpression optimisation. This ensures that the bindings generated for the body of $[\lambda y : y \cdot x \cdot x]$ are $\{1 \mapsto x\}$ instead of $\{1 \mapsto x\ ,\ 2 \mapsto x\}$

The final addition is devoted to generating the combinator body itself. Each time a subexpression c generates a binding $n \mapsto c$, expression c is replaced by a reference to the newly introduced variable: $\mathsf{cid}(n)$.

4.3 Compiling

So far we described properties of the supercombinator code. Now we are ready to discuss the actual compilation of *nexp* to *cexp*. In order to achieve this, we already extended *nexp* with a synthesized attribute of type *cexp*. This attribute will contain the supercombinator code of the underlying *nexp* expression. Compilation of nap, nid, and nco is straightforward, nop still requires some work because the applications to the abstracted expressions have to be computed.

In case of a $\mathsf{nop}(name, nexp{<}\uparrow c{>})$, we must eliminate the λ and introduce a Λ. Hence we must determine the combinator body and bindings of c. This simply means that we have to attribute expression c! Therefore we introduce a non-terminal attribute:

$$
\begin{array}{lll}
\multicolumn{3}{l}{nexp{<}\uparrow cexp{>}} \\
nexp{<}\uparrow c_0{>} & ::= & \mathsf{nop}(name, nexp{<}\uparrow c_1{>}, \overline{cexp}{<}\uparrow l, \downarrow n, \downarrow i, \uparrow o, \uparrow c_2{>}) \\
 & & \overline{cexp} := c_1; \\
 & & n := name;\ i := \{\}; \\
 & & c_0 := \mathrm{fold}(\mathsf{cop}(\Pi_1(a)\mathbin{+\!\!+}[0], c_2), \Pi_2(a)) \text{ where } a = \mathrm{tolist}(o) \\
 & | & \mathsf{nap}(nexp{<}\uparrow c_1{>}, nexp{<}\uparrow c_2{>}) \\
 & & c_0 := \mathsf{cap}(c_1, c_2) \\
 & | & \mathsf{nid}(name) \\
 & & c_0 := \mathsf{cid}(name) \\
 & | & \mathsf{nco}(id) \\
 & & c_0 := \mathsf{cid}(id)
\end{array}
$$

where "tolist" converts a set of bindings to a list of bindings and

$$
\begin{array}{ll}
\mathrm{fold} & :: cexp \rightarrow [cexp] \rightarrow cexp \\
\mathrm{fold}(c, [\,]) & = c \\
\mathrm{fold}(c, m\mathbin{+\!\!+}[a]) & = \mathsf{cap}(\mathrm{fold}(c, m), a) \\
 & \\
\Pi_1 & :: [name \mapsto cexp] \rightarrow [name] \\
\Pi_1([\,]) & = [\,] \\
\Pi_1(o\mathbin{+\!\!+}[n \mapsto c]) & = \Pi_1.o\mathbin{+\!\!+}[n] \\
 & \\
\Pi_2 & :: [name \mapsto cexp] \rightarrow [cexp] \\
\Pi_2([\,]) & = [\,] \\
\Pi_2(o\mathbin{+\!\!+}[(n \mapsto c]) & = \Pi_2.o\mathbin{+\!\!+}[c]
\end{array}
$$

The function "tolist" that converts a set to a list offers a lot of freedom: we may pick any order we want. We may exploit this freedom to generate better code: order the expressions in such a way that their levels are ascending. Since application is left associative this results in the largest maximal free expressions for the surrounding expression.

5 Related formalisms

In this section we will discuss a number of related approaches, trying to solve the kind op problems discussed in this chapter. In the end of this chapter HAGs are positioned between several other programming formalisms, and their strengths and weaknesses will be placed into context.

5.1 ACGs

Attribute Coupled Grammars were introduced in [Ganzinger and Giegerich 84] in an attempt to model the multi-pass compilation process. Their model can be considered as a limited application of HAGs, in the sense that they allow a computed synthesized attribute of a grammar to be a tree which will be attributed again. This boils down to a HAG with the restriction that NTA may be only instantiated at the outermost level.

5.2 EAGs

Extended Affix Grammars [Koster 91] may be considered as a practical implementation of Two-Level Grammars. By making use of the pattern matching facilities in the predicates (i.e. non-terminals generating the empty sequence) it is possible to realise a form of control over a specific tree. The style of programming in this way resembles strongly the conventional Miranda style. An (implicitly) distinguished argument governs the actual computation which is taking place. Extensive examples of this style of formulation can be found in [Cleaveland and Uzgalis 77]. Here you may find an thorough introduction into Two-Level grammars, and as an example a complete description of a programming language, including its dynamic semantics, is given.

5.3 Functional languages with lazy evaluation

It is a well-known secret that attribute grammars may be directly mapped onto lazy-evaluated functional programming languages: the non-terminals correspond to functions, the productions to different parameter patterns and associated bodies, the inherited attributes to parameters and the synthesized attributes to elements of the result record [Kuiper and Swierstra 87].

This mapping depends essentially on the fact that the functional language is evaluated lazy. This makes it possible to pass an argument which depends on a part of the function result. In functional implementations of AGs this seeming circularity is transformed away by splitting the function into a number of functions corresponding to the repeated visits of the nodes. In this way some functional programs might be converted to a form which no longer essentially depends on this lazy evaluation. All parameters in the attribute grammar formalism correspond to strict parameters in the functional formalism because of the absence of circularities.

Most functional languages which are lazy evaluated however allow circularities. In that sense they may be considered to be more powerful.

5.4 Schema

In this section we will try to give a schema which may be used to position different programming formalisms against each other. The basic task to be solved by the different implementations will be to solve a set of equations. As a running example we will consider the following set:

$$\begin{array}{lrcl} (1) & x & = & 5 \\ (2) & y & = & x + z \\ (3) & z & = & v \\ (4) & v & = & 7 \end{array}$$

- **garbage collection (GC)**
  One of the first issues we mention captures the essence of the difference between functional and declarative styles on the one hand and the imperative styles on the other. When solving such a set of equations there may be a point that a specific variable is not occurring any more in the set because it has received a value and this value has been substituted in all the formulae. The location associated with this variable may thus be reused for storing a new binding. In an imperative programming language a programmer has to schedule its solution strategy in such a way that the possibility for reuse is encoded explicitly in the program. An assignment not only binds a value to a variable, but it also destroys the previously bound value, *and thus has the character of an explicitly programmed garbage collection action.* So after substituting x in equation (2), we might forget about x and use its location for the solution of further equations.

- **direction (DIR)**
  The next distinction we can make is whether the equations are always used for substitution in the same direction, i.e. whether it is always the case that the left hand side is a variable which is being replaced by the right hand side in the other equations. This distinction marks the difference between the functional and the logical languages. The first are characterised by exhibiting a direction in the binding, whereas the latter allow substitutions to be bi-directional. Depending on the direction we might substitute (3) and (4) by a new equation $z = 7$ or (2) and (3) by $y = x + v$

- **sequencing (SEQ)**
  Sequencing governs the fact whether the equations have to be solved in the way they are presented, or whether there is still dynamic scheduling involved, based on the dependencies. In the latter case we often speak of a demand driven implementation, corresponding to lazy evaluation; in the first case we speak of an applicative order evaluation, which has a much more restricted scheduling model. In the example it is clear that we cannot first determine the value for x, then y and finally z and v. As a consequence some languages are not capable of handling the above set of equations.

- **dynamic set of equations (DSE)**
  One of the things we have not shown in our equations above is that often we have to do with a recursively defined set of equations or indexed variables. In languages these are often represented by use of recursion in combination with conditional expressions or with loops. We make this distinction in order to distinguish between the normal AGs and the HAGs.

In the table in Figure 10 we have given an overview of the different characteristics of several programming languages. The +'s and −'s are used to indicate the ease of use for a programmer in respect to his programming task, and thus do not reflect things like efficient execution or general availability.

Based on this table we may conclude that HAGs bear a strong resemblance to functional languages like Miranda. Things which are still lacking are infinite data structures, poly-

	GC	DIR	SEQ	DSE
Pascal	−	−	−	+
Lisp	+	−	−	+
Miranda	+	−	+	+
AG	+	−	+	−
HAG	+	−	+	+
Prolog	+	+	+/−	+
Pred. Logic	+	+	+	+

Figure 10: An overview of language properties

morphism, and more powerful data structures. The term structures which are playing such a prominent rôle in attribute grammars are not always the most natural representation.

6 Basic incremental evaluation techniques

In this section the problems with incremental evaluation of HAGs, conventional incremental evaluation techniques for AGs and basic techniques for the incremental evaluation of single visit HAGs will be presented. The incremental evaluation of OHAGs will be discussed in the next section.

6.1 Problems with HAGs

The two main problems in the incremental evaluation of HAGs are the efficient evaluation of multiple instantiations of the same NTA and the incremental evaluation after updating a NTA. In section 2.2 we saw the replacement of a (semantic) lookup-function by a NTA. This NTA then takes the rôle of a semantic function. As a consequence, at all places in an attributed tree were the lookup-function would have been called the (same-shaped) NTA will be instantiated. Such a situation is shown in Figure 11 where T2 is the tree modelling e.g. part of the environment, and is being joined with T3 and T4 giving rise to two larger environments. X1, X2 are the locations in the attributed tree were these two trees are instantiated. These instantiations thus include a copy of the tree T2. The following can be noted with respect to incremental evaluation in Figure 11, where the situation (a) models the state before an edit action in the subtree indicated with NEW, and (b) the situation after the edit action:

- NTA1 and NTA2 are defined by attribution.
- Trees T2 and T'2 are multiple instantiated trees in both (a) and (b). How can we achieve an efficient representation for multiple instantiated (equal or non-equal attributed) trees like T2 and T'2 in (a) and (b)?

- NTA1 and NTA2 are updated when a subtree modification occurs at node NEW. How can we identify efficiently those parts of an attributed tree (like T3 and T4 in (b)) derived from an NTA which can be reused after NTA1 and NTA2 have been updated?

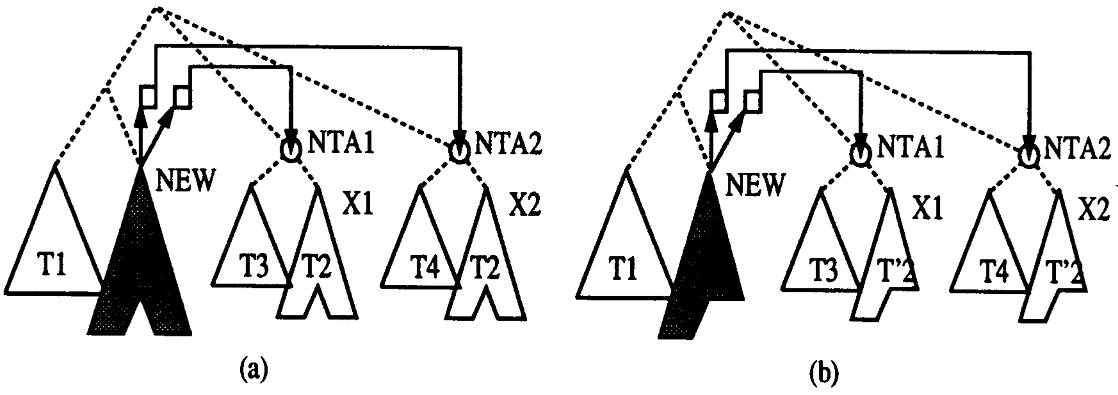


Figure 11: A subtree modification at node NEW induces subtree modifications at node X1 and X2 in the trees derived from NTA1 and NTA2.

Note that these problems do not occur in conventional AGs, but arise through the NTAs in HAGs. Before we continue with some techniques for the incremental evaluation of single visit HAGs some conventional incremental evaluation techniques are presented.

6.2 Conventional techniques

Below several incremental AG-evaluators will be listed. All of them can be trivially adapted for the higher oder case but none of them is capable of efficiently handling multiple instantiations of the same NTA and reusing slightly modified NTAs.

- OAG [Kastens 80, Reps, Teitelbaum and Demers 83]
- Optimal time-change propagation [Reps, Teitelbaum and Demers 83]
- Approximate Topological Ordering [Hoover 86]
- Function caching [Pugh 88]

The following observations hold for all of the above mentioned incremental evaluators:

- Attributes are stored in the tree. The tree functions as a cache for the semantic functions during incremental evaluation.
- Equal structured trees are not shared. This is difficult because the attributes are stored with the tree, and the opportunity for such sharing does not arise too often.

As will be shown later, the above two observations limit efficient incremental evaluation of HAGs.

6.3 Single visit HAGs

In this subsection we will introduce some methods needed for the efficient incremental evaluator explained in the next section. These steps will be explained by first constructing an efficient incremental evaluator for single visit HAGs. We define a single visit HAG to be a subclass of the Ordered HAGs in which there is precisely one visit associated with each production.

6.3.1 Consider a single visit HAG as a functional program

The HAG shown in section 2.2 is an example of such a single visit HAG. The single visit property guarantees that the visitsequences *VS(p)* actually are *visit functions*, mapping the inherited to the synthesized attributes.

6.3.2 Visit function caching/tree caching

The second step we take is the decision to cache the results of the *visit functions* instead of the results of *semantic functions*, as was done in [Pugh 88]. This is more efficient because a cache hit of a visit function means that this visit to (a possibly large) tree may be skipped. Furthermore, a visit function returns the results of several semantic functions at the same time. Note furthermore that we have modeled in this way the administration of the incremental evaluation by using the function caching. No separate bookkeeping in order to determine which attributes have changed and which visits should be performed is necessary.

The implementation of function caching used for caching the visit functions of the functional evaluator 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 by calling

visit_N T inherited_attributes

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

```
function cached_apply(visit_N, T, args) =
    index := hash(visit_N, T, args)
    forall <function, tree, arguments, result> ∈ cache[index] do
        if function = visit_N and EQUAL(tree,T)
          and EQUAL(arguments,args)
        then return result
        fi
    od
    result := visit_N T args
    cache[index] := cache[index] ∪ {<visit_N, T, args, 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 very large structures!) in the pending function call and in a candidate match.

The case of trees in the last problem is solved by using a technique which has become known as hash-consing for trees. When hash-consing for trees is used, the *constructor* functions for trees are implemented in such a way that they never allocate new constructor-cells with the same value as an already existing cell; instead a pointer to that already existing cell is returned. As a consequence all equal subtrees of all structures which are being built up are automatically shared.

Hash-consing for trees can be obtained by using an algorithm such as the one described below (*EQ* tests true equality). As a result hash-consing allows constant-time equality tests for trees.

```
function hash_cons(CONSTR, $(p_1, p_2, \ldots, p_n)$) =
    index := hash(CONSTR, $(p_1, p_2, \ldots, p_n)$)
    forall  p ∈ cache[index] do
        if p^.constructor = CONSTR
            and EQ(p^.pointers, $(p_1, p_2, \ldots, p_n)$)
        then return p
        fi
    od
    p := allocate_constructor_cell()
    p^ := <CONSTR, $(p_1, p_2, \ldots, p_n)$>
    cache[index] := cache[index] ∪ {p}
    return p
```

Now, the function call *EQUAL(tree1, tree2)* in *cached_apply* may be replaced by a pointer comparison (*tree1* = *tree2*) in our previous algorithm. As for function caching, we need an efficient solution for computing a hash index based on a constructor and pointers to memory-cells.

6.4 A large example

Consider again the higher order AG in section 2.2, 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. Figure 12.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 (block   (def(def(def(def empty_decls a) b) c))
                    (use(use(use(use(use empty_apps c) c) b) c)))
```

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

```
visit_ROOT (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 12.b, note that only the APPS-tree will be totally revisited (since the inherited attribute env changed), the first visits to the DECLS and ENV trees generate cache-hits, and further visits to them are skipped.

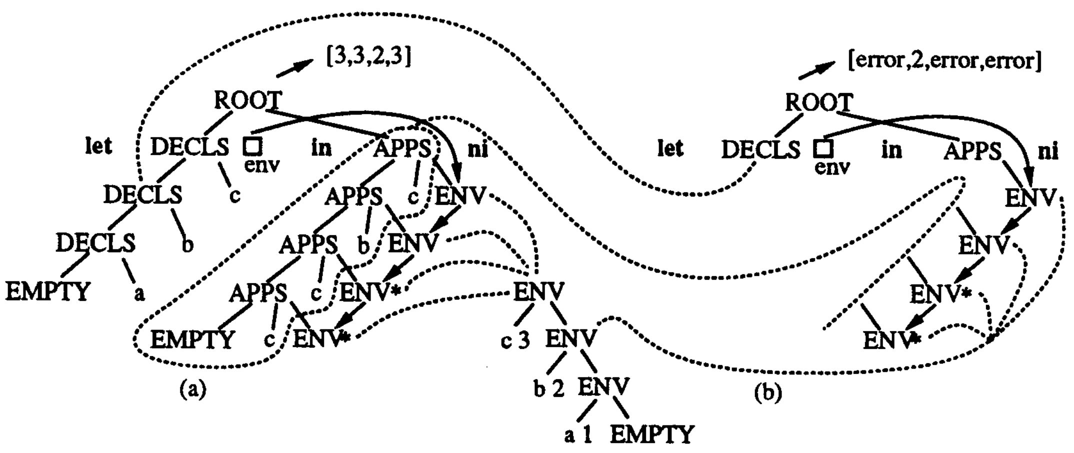


Figure 12: The tree before (a) and after removing c (b) from the declarations in **let** a,b,c **in** c,c,b,c **ni**. The * indicate cache-hits looking up c. The dashed lines denote sharing.

Simulation shows that, when using caching, in this example 75% of all visitfunction calls and tree-build calls which have to be computed in 12.b are found in the cache constructed in evaluation 12.a. So 75% of the "work" was saved. Of course removing a instead of c won't yield the same results.

7 Incremental evaluation of OHAGs

As was shown in the previous section, instead of caching the results of *semantic functions* the results of *visit functions* are cached.

Although this idea seems appealing at first sight, a complication is the fact that attributes computed in an earlier visit have to be available for later visits when necessary and thus the model does not generalise easily to the multi-visit case.

Therefore so called *bindings* are introduced. Bindings contain attribute values computed in one visit and used in one or more subsequent visits to the same tree. So each visit function computes synthesized attributes *and* bindings for subsequent visits. Each visit function will be passed an extra parameter, containing the attribute values which were computed by earlier visits and will be used in this visit. In this sense the implementation bears some resemblance to the super combinator implementation which has been discussed earlier. All the relevant information for the function is being passed explicitly as an argument, and nothing more.

7.1 Informal definition of bindings

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

7.1.1 Visit(sub)sequences

In the section 3 the so called Ordered Higher order Attribute Grammars (OHAGs), have been defined. An OHAG is characterized by the existence of a total order on the defining attribute occurrences in the productions p. This order induces a fixed sequence of computation for the defining attribute occurrences, applicable in any tree node labeled with production p.

Such a fixed sequence is called a *visitsequence* and will be denoted by *VS(p)*. A visitsequence is an ordered list of instructions of the following four types (we will use a slightly different notion of visitsequences as normal): "$\gamma := \ldots$" (evaluate attribute γ, a copy of the attribution rule in the grammar), "$\overline{\gamma} := \ldots$" (evaluate non-terminal attribute $\overline{\gamma}$ and expand the tree with the tree computed in $\overline{\gamma}$, again a copy of the rule in the grammar), *visit_*X_i*_*j (visit son X_i for the j-th time) and *visit_*X_0*_*j (visit the parent for the j-th time). *VS(p)* is split into *visitsubsequences VSS(p,v)* where each visit is terminated by a visit to the parent node. The attribute grammar in Figure 13 will be used in the sequel to demonstrate visitsubsequences, bindings and visit functions.

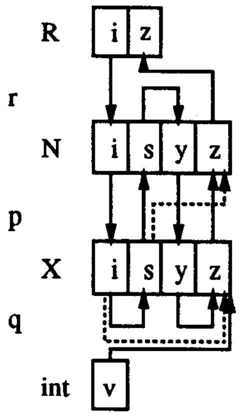
7.2 Visit functions for the example grammar

The evaluator is obtained by translating each visitsubsequence *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 85] 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.

```
R ::= r(N)    { N.i := R.i; N.y := N.s; R.z := N.z }
N ::= p(X)    { X.i := N.i; N.s := X.s; X.y := N.y; N.z := X.z + X.s }
X ::= q(int)  { 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 13: An example AG (top), the dependencies (left) and visitsequences (right). The dashed lines indicate dependencies on an attribute defined in the first visit and used in the second visit. VS(r) is omitted.

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 13 again. The inherited attribute X.i and the synthesized attribute X.s in Figure 13 are also used in the second visit to X and N 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 to N). So *visit_N_v* computes also $nov_N - v$ bindings, one for each subsequent visit (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 $binds\_N^{v \to v+i}$.

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. $binds\_N^{i \to v}.1$ is the first element of the list.

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

```
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, denoted X. The second argument is the inherited attribute i of N. The function returns the synthesized attribute s of N and a binding containing $X.s$ 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. Here the projections on $binds\_N^{1\rightarrow 2}$ can be made implicit by replacing $[binds\_N^{1\rightarrow 2}]$, the last parameter of *visit_N_2*, by $[\boxed{\text{X.s}}, binds\_X^{1\rightarrow 2}]$.

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

The other visit functions have a similar structure.

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

$visit\_X\_2$ (q [int]) X.y [$binds\_X^{1\rightarrow 2}$] = X.z
        where $\boxed{\texttt{X.i}}$ = $binds\_X^{1\rightarrow 2}$.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 $binds\_X^{1\rightarrow 2}$ is looked up in $binds\_N^{1\rightarrow 2}$).
3. definitions for returned bindings. The precise definition of visit functions and bindings is given in section 7.3.

7.2.1 Incremental evaluation

After a tree T is modified into T', 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 rebuild before *visit_R_1* is called, and all parts in T' that are copies of parts in T are identified automatically by the hash consing for trees.

The incremental evaluator automatically skips unchanged parts of the tree because of cache-hits of visit functions. Hash consing for trees and bindings is used to achieve efficient caching, for which fast equality tests are essential. Because separate bindings for each visit are computed, 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 11, in which the following can be noted:

- Multiple instances of the same (sub)tree, for example a multiple instantiated NTA, 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 are 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 $O(|\text{Affected T'2}| + |\text{paths\_to\_roots}|)$ steps, where paths_to_roots is the sum of the lengths of all paths from the root to all subtree modifications (NEW, X1 and X2).

7.3 Definitions of 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 $\rightarrow \dots X_i \dots$. Let *VS(p)* be the visitsequence for p. As before, nov_N is the number of visits to N. Let *VSS(p,1)* … *VSS(p,nov_N)* be the visitsubsequences in *VS(p)*.

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

```
visit_N_v (p [... X ...]) $inh_v^N$ [$binds\_N^{1\rightarrow v}$, ..., $binds\_N^{(v-1)\rightarrow v}$] =
                          ($syn_v^N$, $binds\_N^{v\rightarrow v+1}$, ..., $binds\_N^{v\rightarrow nov_N}$)
                          where Lines from 1) to 3).
                                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,w is translated into

$$\begin{aligned}&(syn_w^X,\ binds\_X^{w\rightarrow w+1},\ \dots,\ binds\_X^{w\rightarrow nov_X}) = \\ &\qquad visit\_X\_w\ X_i\ inh_w^X\ [binds\_X^{1\rightarrow w},\ \dots,\ binds\_X^{(w-1)\rightarrow w}]\end{aligned}$$

When X is a non-terminal attribute, the variable defining X is used as the first argument pattern for *visit_X_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 = $binds\_N^{e \rightarrow v}.f$ is added, for the appropriate e and f.

In 3) the bindings returned by *visit_N_v* are defined. Recall that the $binds\_N^{v \rightarrow v+i}$ are defined in terms of the visitsequence of production p. $binds\_N^{v \rightarrow v+i}$ 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 $inout\_N_p^{v \rightarrow v+i}$) **plus** the bindings of sons of N computed by visits to N and used in subsequent visits to those sons of N in *visit_N_v+i* (denoted by $binds\text{-}sons\_N_p^{v \rightarrow v+i}$). For example $binds\_N^{1 \rightarrow 2}$ in the example visit functions in section 2 is

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

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

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

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

The following theorem holds for the above defined functional program.

Theorem 7.1 *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.*

7.4 Incremental evaluation performance

In this section the performance of the functional evaluator with respect to incremental evaluation is discussed. The goal is to prove that the derived incremental evaluator recomputes in the worst case a number of semantic function calls bounded by $O(|Affected|)$. Here *Affected* is the set of attribute instances in the tree which contain a different value, together with the set of attribute instances newly created after a subtree modification.

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* are all nodes on the path to the initial subtree modification and 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 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 7.3. 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').

Theorem 7.2 *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 visits and hash consing for trees. Then, Affected_Applications is O(|Affected| + |paths_to_roots|).*

8 Discussion

In the foregoing sections we have shown how an incremental evaluator may be based on concepts like hash-consing and function caching. Here we will elaborate on some further possibilities for optimisation.

8.1 Skipping subtrees

An essential property of the construction of the bindings was that when calling a visit function with its bindings, these bindings contain precisely that information that will be actually used in this visit *and no more.* This is a direct result of the fact that these bindings were constructed during earlier visits of the nodes, at which visits it was known what productions had been applied and what dependencies are actually occurring in the subtrees. There is thus little room for improvement here.

The situation is different however when we inspect the rôle of the first parameter to the visit functions more closely: always the complete tree is passed and not only that subtree that will actually be traversed by this visit. In this way we might miss a cache hit when evaluating a changed tree. This effect is demonstrated in Figure 14. When editing the shaded subtree this has no influence on the outcome of pass b, and may only influence pass a.

The following modification of our approach will take care of this optimisation. When building the tree we compute simultaneously those synthesized attributes of the tree which do not depend on any of the inherited attributes. In this process we also compute a set of functions which we return as synthesized attributes, representing the visit functions parameterised with that part of the tree which will be visited when they are called.

This process consists of the following steps:

1. Every visit corresponds to a visit function definition. At those places where the visit subsequences contain visits to sons, a formal function is called. Each visit function thus has as many additional parameters as is contains calls to sons.

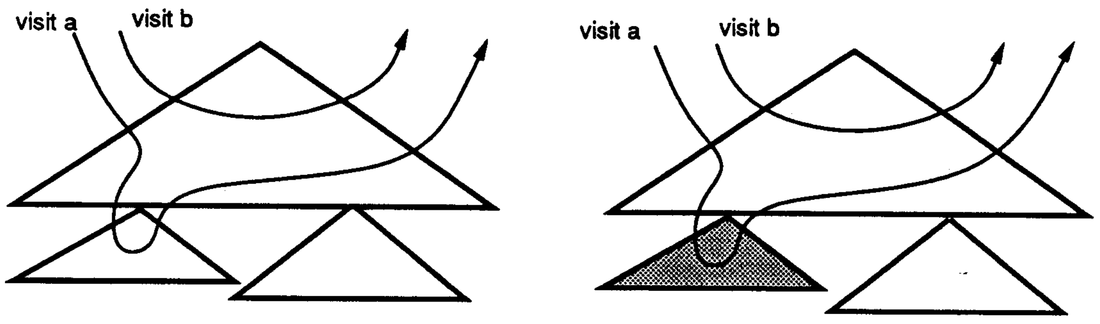


Figure 14: Changes in an unvisited subtree

2. The synthesized attributes computed initially represent functions representing the calls to the subtrees. These functions are used to partially parameterize the visit functions definitions associated with production applied at the current node under construction, and these resulting applications are in their turn passed to higher nodes via the synthesized attributes.

As a consequence of this approach the top node of a tree is represented by a list of visit functions, all partially parameterized by the appropriate calls to their sons. Precisely *those parts of the trees which are actually visited* by these functions are thus encoded via the partial parameterisation. If the function cache is extended in such a way as to be able to distinguish between such values, we do not have to build the trees at all, and may simply use the visit functions as a representation.

8.2 Removing copy rules

As a final source for improvement we have a look at a more complicated case where we have visits which pass through different, but not distinct parts of the subtree. An example of this is the case were we model a language which does not demand identifiers to be declared before they may be used. This naturally leads to a two-pass algorithm: one pass for constructing the environment and the second pass for actually compiling the statements.

We will base our discussion on the tree in Figure 15. We have indicated the dataflow associated with the computation of the environment as a closed line, and the data flow of the second pass which actually computes the code with a dashed line. Notice that the first line passes through all the declaration nodes, whereas the second line passes through all the statement nodes.

Suppose now that we change the upper statement in the tree, and thus construct a new root. If we apply the aforementioned procedure, we will discover that we do not have to redo the evaluation of the environment. The function computing this environment has not changed.

The situation becomes more complicated if we add another statement after the first one.

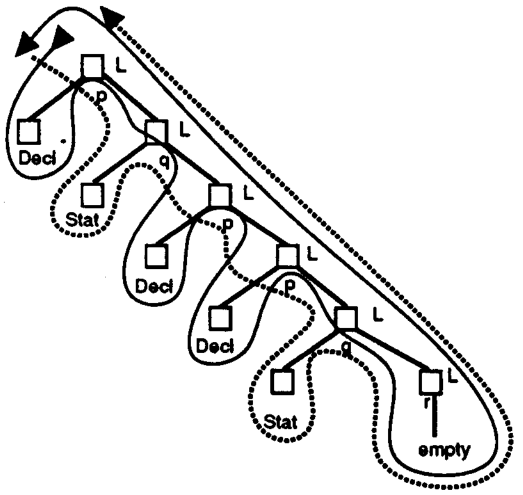


Figure 15: Removing copy rules

Strictly speaking this does not change the environment either. However the function computing the environment has changed, and will have to be evaluated anew. This situation may be prevented by noticing the following. The first visit to an L-node which has a statement as left son actually passes the environment attribute to its right son, visits this right son for the first time and passes the result up to its father. No computation is performed. When writing this function, with the aforementioned transformation in mind, as a λ-term we get $\lambda f, x : f(x)$, where f represents the visit to the right son, and x the environment attribute. When we partially parameterize this function however with a function g, representing the visit to the right son, this rewrites to $\lambda x : g(x)$, which is equal to g. In this way copy-chains may be short-circuited and the number of cache hits may increase by making more functions constructed this way to be equal. Consider, as an example, the first pass visit functions for the grammar of Figure 15:

```
visit_L_1 (p [D,L]) env = L.env
        where D.env = visit_D_1 D env
              L.env = visit_L_1 L D.env
visit_L_1 (q [S,L]) env = L.env
        where { S contains no declarations }
              L.env = visit_L_1 L env
```

The visit functions for production p may be shortcircuited to

```
visit_L_1 (p [D,(q [S,L²])]) env = L.env
        where D.env = visit_D_1 D env
              { the copyrules for S may be skipped }
              L.env = visit_L_1 L D.env
```

[2]These visit functions are merely meant to sketch the idea. In case L=(q [S2,L2]), we may shortcircuit two statement nodes (and so on). This is what the aforementioned transformation is about.

```
visit_L_1 (p [D1,(p [D2,L])]) env = L.env
       where D1.env = visit_D_1 D1 env
             L.env = visit_L_1 (p [D2,L]) D1.env

visit_L_1 (p [D,(r [empty])]) env = L.env
       where L.env = visit_D_1 D env
```

We conclude by noticing that whether these optimisations are possible or not depends on the amount of effort one is willing to spend on analysing the grammar, reordering attributes, and splitting up visits into smaller visits. The original visit functions of [Kastens 80] were designed with the goal to minimise the number of visits to each node in mind. In the case of incremental evaluation one's goals however will be to maximise the number of independent computations and to maximise the number of cache hits.

Acknowledgements

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

References

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

[Cleaveland and Uzgalis 77] J. Craig Cleaveland and Robert C. Uzagalis Grammars for Programming Languages Elsevier/North-Holland 1977.

[Ganzinger and Giegerich 84] Ganzinger, H., and R. Giegerich. Attribute Coupled Grammars *Sigplan Notices Vol. 19, No. 6*, pages 157–170, 1984.

[WAGA 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.

[Hoover 86] Hoover, R. *Dynamically Bypassing Copy Rule Chains in Attribute Grammars.* In Proc. of the 13th ACM Symp. on Principles of Programming Languages, St. Petersburg, FL, Jan 13-15, pages 14-25, 1986.

[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.

[Hughes 85] Hughes, R.J.M. *Lazy memo functions.* In Proc. Conference on Functional Progr. and Comp. Architecture, Nancy, pages 129–146, LNCS 201, Springer Verlag, 1985.

[Jazayeri 1975] Jazayeri, M., W.F. Ogden, W.C. Rounds. *The intrinsically exponential complexity of the circularity problem for attributed grammars.* In CACM 18, pages 679–706, 1975.

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

[Knuth 68] Knuth, D.E. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127–145, 1968.

[Knuth 71] Knuth, D.E. Semantics of context-free languages (correction). *Math. Syst. Theory*, 5(1):95–96, 1971.

[Koster 91] Koster, C.H.A. Affix Grammars for Programming Languages this summer school

[Kuiper and Swierstra 87] Kuiper, M.F. and S.D. Swierstra *Using Attribute Grammars to Derive Efficient functional programs.* Computing Science in the Netherlands, SION, November 1987.

[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 82] Reps, T. *Generating language based environments.* Tech. Rep. 82-514 and Ph.D dissertation, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Aug. 1982.

[Reps, Teitelbaum and Demers 83] Reps, T. , T. Teitelbaum and A. Demers. *Incremental Context-Dependent Analysis for Language Based Editors.* In ACM Transactions on Progr. Lang. and Systems, Vol. 5, No. 3, pages 449-477, July 1983.

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

[TC90] 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 85] 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.

[Turner 79a] Turner, D.A., *A New Implementation Technique for Applicative Languages* In *Software, Practice and Experience*, vol. 9, pages 31–49, 1979.

[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.

[Vogt, Swierstra and Kuiper 91] Vogt, H.H., S.D. Swierstra and M.F. Kuiper. *Efficient incremental evaluation of higher order attribute grammars.* In the proceedings of the International Symposium on Programming Language Implementation and Logic Programming, Passau (FRG), August 26-28, (To Appear), 1991.

[Waite and Goos 84] Waite, W.M. and G.Goos. *Compiler Construction.* Springer, 1984.

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