Functional Incremental Attribute Evaluation

João Saraiva[1], Doaitse Swierstra[2], and Matthijs Kuiper[3]

[1] Depart. of Informatics
University of Minho, Portugal
jas@di.uminho.pt
[2] Depart. of Computer Science
University of Utrecht, The Netherlands
swierstra@cs.uu.nl
[3] Ordina, The Netherlands
MKuiper@factory.ordina.nl

Abstract. This paper presents a new strict, purely functional implementation of attribute grammars. Incremental evaluation is obtained via standard function memoization. Our new implementation of attribute grammars increases the incremental behaviour of the evaluators by both reducing the memoization overhead and increasing their potential incrementallity. We present also an attribute grammar transformation, which increases the incremental performance of the attribute evaluators after a change that propagates its effects to all parts of the syntax tree. These techniques have been implemented in a purely functional attribute grammar system and the first experimental results are presented.

1 Introduction

Developments in programming languages are changing the way in which we construct programs: naive text editors are now replaced by powerful programming language *environments* which are specialized for the programming language under consideration and which help the user throughout the editing process. They use *knowledge* of the programming language to provide the users with powerful mechanisms to develop their programs. This knowledge is based on the *structure* and the *meaning* of the language. To be more precise, it is based in the syntactic and (static) semantic characteristics of the language. Having this knowledge about a language, the language-based environment is not only able to highlight keywords and beautify programs, but it can also detect features of the programs being edited that, for example, violate the properties of the underlying language. Furthermore, a language-based environment may also give information to the user about properties of the program under consideration. Finally, it may hide from the user some peculiarities included in the language that are relevant only for the programs/computers that process/execute them. All these features make a language-based environment an effective mechanism to increase the productivity of users.

After each interaction with the user a language-based environment provides immediate feedback concerning whether or not such an interaction is legal. That

A. Watt (Ed.): CC/ETAPS 2000, LNCS 1781, pp. 279–294, 2000.
© Springer-Verlag Berlin Heidelberg 2000

is to say, the environment has to react and to provide answers in real-time. Consequently, the delay between the user interaction and the system response is an extremely important aspect in such interactive systems. Thus, one of the key features to handle such interactive environments is the ability to perform efficient re-computations. Attribute grammars, and particularly their higher-order variant, are a suitable formalism to specify such language-based environments. Furthermore, attribute grammar are executable, *i.e.*, efficient implementations are automatically obtained.

In this paper we introduce a new strict, purely functional implementation for ordered attribute grammars [Kas80]. A strict model of attribute evaluation is attractive for two main reasons: firstly, because we obtain very efficient implementations in terms of memory and time consumption. Secondly, because a rather efficient and simple incremental attribute evaluator can be derived from an attribute grammar: incremental evaluation is obtained via standard function memoization. Our new functional implementation improves the incremental behaviour of the evaluators in two ways: first, by dynamically specializing the underlying syntax tree for each of the individual traversal functions, which increases the sharing of subtrees and consequently the reuse of their shared decorations. Second, by reducing the interpretative overhead due to the memoization scheme: our technique does not induce any additional parameter/result to the evaluators functions which "glue" the different traversals. Such parameters have to be tested for equality under the memoization scheme. Consequently, fewer arguments means fewer equality tests, and as a result, lower interpretative overhead. A change that propagates its effects to all parts of the syntax tree is known to give poor performance in all models of incremental attribute evaluation. In this paper we define a simple attribute grammar transformation that *projects attributes* and that greatly improves their incremental behaviour when re-decorating the syntax tree after such a change.

In Section 2 we discuss incremental attribute evaluation, higher-order attribute grammars and the visit-function memoization scheme. In Section 3 our new functional attribute evaluator is introduced and is briefly compared with previous approaches. Section 4 presents the grammar transformation. Section 5 presents the first experimental results and Section 6 contains the conclusions.

2 Incremental Attribute Evaluation

Attribute grammars (AG) have been used with great success in the development of language-based tools since Thomas Reps first used attribute grammars to model syntax-oriented editors (in Figure 2 we will show a traditional language-based editor). In such an interactive programming environment, a user slightly modifies a *decorated tree* `T` into `T'` (to be more precise, the user changes a "pretty printed" version of such trees). After that, an *incremental attribute evaluator* uses `T` and its attributes instances to compute the attributes instances of `T'`, instead of decorating `T'` from scratch. The underlying assumption is that the

decoration of `T'` from scratch is more expensive (*i.e.*, more time consuming) than an incremental update of `T`.

Although any non-incremental attribute evaluator can be applied to completely re-decorate tree `T'`, the goal of an *optimal incremental attribute evaluator* is to limit the amount of work to $\mathcal{O}(|\Delta|)$, where Δ is the set of *affected* and *newborn* attribute instances. The traditional approach to achieve incremental evaluation involves *propagating changes* of attribute values through the attributed tree [RTD83,RT89].

2.1 Higher-Order Attribute Grammars

Higher-Order Attribute Grammars (HAG) [VSK89] are an important extension to the attribute grammar formalism. Conventional attribute grammars are augmented with *higher-order attributes*. Higher-order attributes are attributes whose value is a tree. We may associate, once again, attributes with such a tree. Attributes of these so-called *higher-order trees*, may be higher-order attributes again. Higher-order attribute grammars have three main characteristics:

- First, when a computation can not be easily expressed in terms of the inductive structure of the underlying tree, a better suited structure can be computed before. Consider, for example, a language where the abstract grammar does not match the concrete one. Consider also that the semantic rules of such a language are easily expressed over the abstract grammar rather than over the concrete one. The mapping between both grammars can be specified within the higher-order attribute grammar formalism: the attribute equations of the concrete grammar define a higher-order attribute representing the abstract grammar. As a result, the decoration of a concrete syntax tree constructs a higher-order tree: the abstract syntax tree. The attribute equations of the abstract grammar define the semantics of the language.
- Secondly, semantic functions are redundant. In higher-order attribute grammars every computation can be modelled through attribution rules. More specifically, inductive semantic functions can be replaced by higher-order attributes. For example, a typical application of higher-order attributes is to model the (recursive) lookup function in an environment. Consequently, there is no need to have a different notation (or language) to define semantic functions in AGs.
- The third characteristic is that part of the abstract tree can be used directly as a value within a semantic equation. That is, grammar symbols can be moved from the syntactic domain to the semantic domain.

These characteristics make higher-order attribute grammars particularly suitable to model interactive language-based environments [TC90,Pen94,KS98,Sar99]. It is known that the incremental attribute evaluator for ordered attribute grammars [Kas80,RT89] can be trivially adapted for the incremental evaluation of higher-order attribute grammars. The adapted evaluator, however, decorates every instance of a higher-order attribute separately [TC90]. Note that in such traditional evaluators the usual representation

for an attributed tree is an *unshared tree*. Higher-order attribute grammars define higher-order attributes, which instances are higher-order trees. Different instances of higher-order attributes may be equal trees or share common subtrees. Consequently, the most efficient representation for such high-order trees is a shared tree, *i.e.*, a *directed acyclic graph* (DAG). Consequently, there is a clash between the two representations, namely, tree and DAG. There are two ways to solve this tension: either we use specific techniques to decorate DAGs or we guarantee that the terms are, in fact, trees that are decorated by an adapted evaluator.

Let us discuss first the use of adapted evaluators. Teitelbaum [TC90] proposes a simple approach to handle DAGs: whenever a higher-order attribute has to be instantiated with a higher-order tree (*i.e.*, with a DAG), the tree sharing is broken and the attribute is actually instantiated with a tree. This tree is a "*tree-copy*" of the DAG. After that, the higher-order tree can be easily decorated by an adapted *change propagator* since attribute values can be associated with the tree nodes in the standard way. This approach, however, leads to a non-optimal incremental behaviour when higher-order attributes are affected by tree transformations, as shown in [CP96]. Note that as a result of breaking the sharing, different instantiations of higher-order attributes are, indeed, different trees. Such trees have to be decorated separately, without the possibility of reusing attribute values across the different decorations of those attributed trees. Note that instances of the same higher-order attribute are likely to be (completely or partially) decorated with the same attribute values. In order to efficiently (and incrementally) evaluate such instances, the reuse of those values should be achieved.

2.2 Visit-Function Memoization

The visit-function memoization proposed by Pennings [Pen94] is based on the following combination of ideas:

Purely functional attribute evaluators: Syntax trees are decorated by *binding-tree based attribute evaluators* [Pen94]. Such evaluators are strict, purely functional attribute evaluators. Attribute instances are not stored in the tree nodes, but, instead, they are the arguments and the results of (side-effect free) functions: the *visit-functions*. Binding-trees are used with the single purpose to "glue" the different traversal functions of the evaluator, *i.e.*, they convey information between traversals.

Data constructor memoization: Since attribute values are not stored in the syntax tree, multiple instances of the syntax tree can be shared. That is, trees are collapsed into minimal *direct acyclic graphs*. DAGs are obtained by constructing trees bottom-up and by using constructor memoization to eliminate replication of common sub-expressions. This technique, also called *hash-consing*, guarantees that two identical objects share the same records on the heap, and thus are represented by the same pointer.

The basic idea of hash-consing is very simple: whenever a new node is allocated (or *"consed"*) we check whether there exists already an identical record in the heap. If so, we avoid the allocation and simply use the existing one. Otherwise, we perform the allocation as usual. Generally, a *hash table* is used to search the heap for a duplicated record. Hash-consing can be applied to pure values only, *i.e.*, values that never change during the execution of a program: for if two updatable records have identical values now, they might not be identical later, and so merging them could lead to incorrect values. Observe that this is the case if attributes are stored in the tree nodes, because shared nodes may have to store different attribute values induced by different, but shared, attribute instances (and most probably will).

Attribute values may be large structures (*e.g.*, symbol tables, higher-order attributes, etc). Therefore, the constructors for user defined types are also shared. This technique solves the problem of expensive attribute equality test during evaluation and it also settles the problem of huge memory consumption due to multiple instances of the same attribute value in a syntax tree.

Function memoization: Due to the pure nature of the visit-functions, incremental evaluation can now be obtained by memoizing calls to visit-functions. The binding-tree based attribute evaluators are constructed as a set of strict functions. Thus, standard function memoization techniques can be used to memoize their calls. Memoization is obtained by storing in a *memo table* calls to visit-functions. Every call corresponds to a *memo entry*, in the *memo table*, that records both the arguments and the results of one call to a visit-function.

Let us describe the binding-tree attribute evaluators in more detail. Such evaluators are based on the visit-sequence paradigm [Kas80]: the visit-sequences induce a set of *visit-functions*, each of them performing the computations schedule for a particular traversal of the evaluator. The different traversal functions are "glued" by intermediate data structures: the *binding-trees*. The visit-functions and the binding-tree data types are automatically induced by a *binding analysis* of the visit-sequences [Pen94].

As a result of the binding analysis, a visit-function $\mathtt{visit}_v\mathbf{X}$ is constructed for every terminal X and every visit v such that $1 \leq v \leq n$, with n the number of visits to X. The arguments of the visit-function are the subtree labelled X, the inherited attributes of visit v and the binding-trees computed in previous visits that are destructed by visit v. The results are the binding-trees for following visits and the synthesized attributes of visit v. The first visit does not inherit any binding-tree, and the last one does not synthesize them. The introduction of binding-trees is reflected in the types of the visit-functions: they are additional arguments/results.

$$\mathtt{visit}_1\mathbf{X} :: X \rightarrow \text{"}A_{inh_v}(X,1)\text{"} \rightarrow (\mathrm{X}^{1\rightarrow 2}, \ldots, \mathrm{X}^{1\rightarrow n}, \text{"}A_{syn_v}(X,1)\text{"})$$
$$\mathtt{visit}_k\mathbf{X} :: X \rightarrow \text{"}A_{inh_v}(X,k)\text{"} \rightarrow \mathrm{X}^{1\rightarrow k} \rightarrow \cdots \rightarrow \mathrm{X}^{k-1\rightarrow k} \rightarrow (\mathrm{X}^{k\rightarrow k+1}, \ldots, \mathrm{X}^{k\rightarrow n}, \text{"}A_{syn_v}(X,k)\text{"})$$
$$\mathtt{visit}_n\mathbf{X} :: X \rightarrow \text{"}A_{inh_v}(X,n)\text{"} \rightarrow \mathrm{X}^{1\rightarrow n} \rightarrow \cdots \rightarrow \mathrm{X}^{n-1\rightarrow n} \rightarrow (\text{"}A_{syn_v}(X,n)\text{"})$$

where $A_{inh_v}(X, i)$ ($A_{syn_v}(X, i)$) denotes the set of inherited (synthesized) attributes that are schedule to be used (computed) during visit i. The quotes around the set of inherited and synthesized attributes of a particular visit should be interpreted as the types of the elements of those sets. Superscripts are used to denote the origins and the targets of binding-trees. Thus, $\mathrm{X}^{v \to w}$ denotes the constructor type of a binding-tree with origin in traversal v ($1 \le v \le n$) and target in traversal w ($v < w \le n$). The original syntax tree does not change during the different traversals of the evaluator. Attribute values are passed between traversals, as arguments and as results of visit-functions, or within binding-trees. Consider the following simplified visit-sequence for production named $\textsc{Prod}\ :\ X \to\ \ Y\ Z$.

```
plan Prod
begin 1 inh(X.inh1)
  visit (Y,1)
  eval  (Y.inh1)   uses(X.inh1, Y.syn1),
  visit (Y,2)
  eval  (X.syn1)   uses(Y.syn2),
end 1   syn(X.syn1)

begin 2 inh(X.inh2)
  visit (Z,1)
  eval  (X.syn2)
        uses(X.inh1, Z.syn1)
end 2   syn(X.syn2)
```

According to this visit-sequences, the inherited attribute $X.inh_1$ must be explicitly passed from the first visit to X (where it is defined) to the second one (where it is used). The nonterminal Y is visited twice, with both visits occurring in the first visit to X. Next, we present the memoized visit-functions (using the Haskell notation[1]) induced by this visit-sequence. Productions' name are used as constructor functions.

```
visit1X (Prod tY tZ) [tXinh1] = (Prod^{1→2} [tXinh1], tXsyn1)
 where (tY^{1→2}, tYsyn1) = memo visit1Y  tY^1
       tYinh1 = f (tXinh1, tYsyn1)
       tYsyn2 = memo visit2Y  tY  tYinh1 tY^{1→2}
       tXsyn1 = tYsyn2
                                  [tXinh1] defined in visit1X
                                           used in    visit2X
visit2X (Prod tY tZ) tXinh2 (Prod^{1→2} [tXinh1]) = tXsyn2
 where tZsyn1 = memo visit1Z  tZ tXinh2
       tXsyn2 = g ([tXinh1], tZsyn1)
```

The function memoization efficiently handles higher-order attribute grammars. Furthermore, the original attribute evaluator needs virtually no change: it does not need any specific purpose algorithm to keep track of which attribute values are affected by a tree change. The memoization mechanism takes care of the incremental evaluation. However, the binding-tree based function memoization has two main drawbacks. Firstly, the memoization mechanism has an interpretative overhead: aiming at improving the potential incremental behaviour of the evaluators, Pennings proposes the use of binding-trees that are *quadratic* in the number of traversals. Although this approach yields binding-tree evaluators which maximize the number of cache hits, such behaviour comes at a price:

[1] The **memo** annotation that we use corresponds to a primitive function of the gofer system [M.P94] that was extended with a (lazy) memoization mechanism [vD92].

first, many binding-tree constructor functions may have to be memoized, which may fill the *memo table*, and, consequently, increase the overhead of its search operation [AG93]. Second, the binding-trees induce additional arguments (and results) to the visit-functions. In our simple example, the functions $\texttt{visit}_2\mathbf{X}$ and $\texttt{visit}_2\mathbf{Y}$ get an additional argument: the binding-trees. When a large number of traversals is induced by the AG scheduling algorithm, then the visit-functions may be extended with a large number of arguments. That is to say, having more binding-trees means having more (argument) values to test for equality, when looking for a memoized call in the *memo table*. Although the equality test among "hash-consed" binging-trees per se is cheap, a large number of tests leads to a not negligible impact in searching the *memo table*.

Secondly, no total traversal of a binding-tree evaluator can be avoided after a tree transformation: the underlying syntax tree changes and, consequently, *all* the visit-functions applied to the newly created nodes (*i.e.*, the nodes in the path from the root to the modified subtrees) have to be (re)computed. The first argument of all the visit-functions that perform the different traversals has changed and, so, no previous calls can be found in the *memo table*.

3 The Visit-Tree Based Attribute Evaluators

In this section we present a new approach for purely functional implementation of attribute grammars that overcomes the two drawbacks presented by the binding-tree approach. First, it reduces the interpretative overhead due to the memoization scheme by not inducing additional "gluing" arguments to the visit-functions. Second, the syntax trees that guide our evaluators are *dynamically specialized* for each visit of the evaluator. This feature allows for entire traversals of the evaluator to be skipped: the traversal functions whose underlying specialized trees do not refer to changed subtrees can be reused.

Basically, our new approach, called visit-tree based attribute evaluator, mimics the imperative approach: attribute values defined in one traversal and used in the following ones are stored in a *new tree*, the so-called *visit-tree*. Such values have to be preserved in the (visit-)tree from the traversal that creates them until the last traversal that uses them. Each traversal builds a new visit-tree for the next traversal, with the additional values stored in its nodes. The functions that perform the subsequent traversals find the values they need, either in their arguments or in the nodes of the (visit-)tree, exactly as in the imperative approach. A set of visit-tree types is defined, one per traversal. A visit-tree for one traversal, say v, is specialized for that particular traversal of the evaluator: it contains only the attribute values which are really needed in traversal v and the following ones. The visit-trees are constructed dynamically, *i.e.*, during attribute evaluation. Consequently, the underlying data structure which guides the attribute evaluator is not fixed during evaluation. This dynamic construction/destruction of the visit-trees allows for an important optimization: subtrees that are not needed in future traversals are *discarded* from the visit-trees concerned. As result, any data no longer needed, no longer is referenced.

The formal derivation of visit-tree based attribute evaluators from higher-order attribute grammars is presented in [Sar99]. In this paper we present the visit-tree approach by informally analysing the production PROD and its visit-sequence.

Let us consider again the attribute $X.inh_1$. In the visit-tree approach, the value of attribute $X.inh_1$ is passed to the second traversal in one visit-tree. This visit-tree is one result of the first traversal of the evaluator. So, we have to define two tree data types: one for the first traversal, and another for the second one. These data types, called *visit-tree data types*, are defined next. We use superscripts to distinguish the traversal they are intended to.

$$\texttt{data}\ \mathrm{X}^1 = \mathrm{PROD}^1\ \ \mathrm{Y}^1\ \ \mathrm{Z}^1$$
$$\texttt{data}\ \mathrm{X}^2 = \mathrm{PROD}^2\ \ \text{“}X.inh_1\text{”}\ \ \mathrm{Z}^1$$

In the first visit to X, the nonterminal Y is visited twice. The visit-tree data type X^1 includes a reference to the first visit (the visit-tree type Y^1) only. The second visit-tree is constructed when visiting the first one (see the visit-functions below). Thus, a reference to the earliest visit suffices. Nonterminal Y is not used in the second visit to PROD. So, no reference to Y has to be included in X^2. Consequently, subtrees labelled Y are discarded from the second traversal of the evaluator. The visit-trees are arguments and results of the visit-functions. Thus, the visit-functions must explicitly destruct/construct the visit-trees.

$$\begin{array}{l}
\texttt{visit}_1\mathbf{X}\ (\mathrm{PROD}^1\ tY^1\ tZ^1)\ \boxed{tXinh_1} = (\mathrm{PROD}^2\ \boxed{tXinh_1}\ tZ^1, tXsyn_1)\\
\ \texttt{where}\ (tY^2, tYsyn_1) = \textbf{memo}\ \texttt{visit}_1\mathbf{Y}\ \ tY^1\\
\qquad tYinh_1 = f\,(tXinh_1, tYsyn_1)\\
\qquad tYsyn_2 = \textbf{memo}\ \texttt{visit}_2\mathbf{Y}\ \ tY^2\ \ tYinh_1\\
\qquad tXsyn_1 = tYsyn_2
\end{array}$$

$\boxed{tXinh_1}$ defined in $\texttt{visit}_1\mathbf{X}$ used in $\texttt{visit}_2\mathbf{X}$

$$\begin{array}{l}
\texttt{visit}_2\mathbf{X}\ (\mathrm{PROD}^2\ \boxed{tXinh_1}\ tZ^1)\ tXinh_2 = tXsyn_2\\
\ \texttt{where}\ tZsyn_1 = \textbf{memo}\ \texttt{visit}_1\mathbf{Z}\ \ tZ^1\ tXinh_2\\
\qquad tXsyn_2 = g\ (\boxed{tXinh_1}, tZsyn_1)
\end{array}$$

tY^2 visit-tree constructed in the first traversal and used in the second.

In the first traversal the attribute value $tXinh_1$ is computed, and a visit-tree node PROD^2 (which stores this value) is constructed. In the second traversal, that node is destructed and the value of $tXinh_1$ is ready to be used. The visit-function $\texttt{visit}_1\mathbf{Y}$ returns the visit-tree for the next traversal (performed by $\texttt{visit}_2\mathbf{Y}$). Observe that the visit-functions $\texttt{visit}_2\mathbf{X}$ and $\texttt{visit}_2\mathbf{Y}$ do not get any gluing arguments. The (visit-)tree and the inherited attributes are the only arguments of the functions.

In order to compare the visit-tree based evaluators with the binding-tree ones, we shall present the induced types of the visit-functions. For each traversal v, with $1 \leq v \leq n$, of a nonterminal symbol X, a visit-function $\texttt{visit}_v\mathbf{X}$ is generated. Its arguments are a visit-tree of type X^{v}, and an additional argument for each attribute in $A_{inh_v}(X, v)$. The result is a tuple, whose first element is the visit-tree for the next traversal and the other elements are the attributes in $A_{syn_v}(X, v)$. The visit-function that performs the last traversal n does not

construct any visit-tree. So, the visit-functions have a signature of the following form:

$$\texttt{visit}_v\mathbf{X} :: \mathrm{X}^{\mathrm{v}} \rightarrow \text{"}A_{inh_v}(X,v)\text{"} \rightarrow (\mathrm{X}^{\mathrm{v}+1}, \text{"}A_{syn_v}(X,v)\text{"})$$
$$\texttt{visit}_n\mathbf{X} :: \mathrm{X}^{\mathrm{n}} \rightarrow \text{"}A_{inh_v}(X,n)\text{"} \rightarrow (\text{"}A_{syn_v}(X,n)\text{"})$$

As expected no additional arguments are included in the visit-functions of our evaluators. The visit-tree based attribute evaluators have the following properties:

- The number of visit-trees is linear in the number of traversals.
- No additional arguments/results in the visit-functions are used to explicitly pass attribute values between traversal functions.
- The visit-functions are strict in all their arguments, as a result of the order computed by the AG ordered scheduling algorithm. Thus, standard function memoization techniques can be used to achieve incremental evaluation.
- Efficient memory usage: data not needed is no longer referenced. References to grammar symbols and attribute instances can efficiently be discarded as soon as they have played their semantic role.

3.1 Combining the Binding and Visit-Tree Approaches

Although the visit-tree approach solves some of the problems of the binding-tree approach, in some situations, it can yield less efficient incremental evaluators than the binding-tree one. This situation occurs when values of attribute instances that have to be passed to following traversals are affected by tree transformations. Recall that under the visit-tree approach these attribute values remain in the tree from when they are created until their last use. Consequently, unnecessary intermediate visits may have to be performed, because the visit-tree is used to pass on such changed values.

To be more precise, let `T'` be an abstract syntax tree resulting from a tree replacement at node `N` in tree `T`. Without loss of generality, consider that a strict evaluator performs three traversals to decorate `T'`. Consider also that an attribute instance α is affected by the tree replacement and that α is defined in the first traversal and used in the third one only. Under the visit-tree approach no traversal of the evaluator can be skipped: *all* the visit-trees of the evaluator are affected by the tree replacement, since all the trees store the changed instance α. Although the second traversal may not be "directly" affected by the change, the visit-functions applied to the nodes in the path from the root to the node where α is stored have to be recomputed because the visit-tree for this traversal has changed. This is represented graphically in Figure 1.

Under the binding-tree approach, the binding-tree makes a "bridge" from the first (origin) to the third (destination) traversal, in order to pass the value of attribute α. As a result, the instance α does not force the re-decoration of part of the second traversal of the evaluator. Nevertheless, part of this traversal has to be decorated because the syntax tree `T` has changed.

We can also improve the potential incremental behaviour of our evaluators by combining visit-trees with binding-trees. The basic idea is that some values are

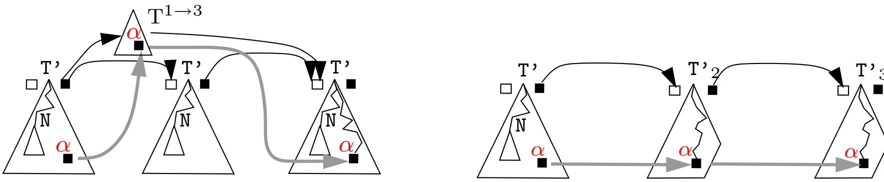

Fig. 1. Passing an attribute value between two non-consecutive traversals: the binding-trees (left) make a "*bridge*" between the origin and the destination traversal. The visit-tree (right) passes the attribute values "*through*" the intermediate traversal.

passed to the following traversals in visit-trees, and others, in binding-trees. Let us be more precise: during visit v of the evaluator, the attribute values that are alive in v and are used in $v+1$ are passed in a visit-tree. Values alive in visit v and which are used in visit w, with $w > v+1$ are passed in a binding-tree. That is, the binding-trees make the bridge for values used in non-consecutive traversals. For example, this approach will efficiently handle the situation presented in Figure 1.

3.2 Semantic Function Memoization

Pugh [PT89] proposes the memoization of the semantic function calls only. In a traditional non-functional setting Pugh's semantic function memoization has the problem of storing attributes in the tree nodes and, consequently, there is no possibility of having tree sharing. Consequently, under a traditional implementation of AGs, Pugh's approach does not handle HAGs efficiently. Besides, in a functional setting, the visit-function memoization is more efficient since the reuse of a visit-function call means that an entire visit to an arbitrarily large tree can be skipped. Such incremental behaviour is not possible under Pugh's approach because all the visits of the evaluator always have to be performed. Nevertheless, Pugh's approach can be easily implemented with the visit-function memoization if one memoizes the calls to semantic functions exactly in the same way as visit-functions.

4 Projection of Attributes

A change that propagates its effects to all parts of the tree causes inefficient behaviour in all the models of incremental atribute evaluation [Pen94,SKS97]. Nevertheless, incremental evaluation can be enhanced greatly by performing a simple transformation on the AG: *the projection of attributes*. Consider, for example, a block structured language. Typically, every inner block inherits the context of its outer block, so, any small change in that context requires the redecoration of the inner blocks, regardless of the irrelevance of the change, *i.e.*,

even if the change is confined to a symbol that is not mentioned in the inner blocks. However, if every block synthesizes a list of used variables, the inherited context could be *projected* on that list, yielding better incremental behaviour.

Next we present an AG fragment defining the projection of the environment in a block structued language. We use a *standard* AG notation: productions are labelled for future references and subscripts are used to distinguish different nonterminal occurrences. Inherited (synthesized) attributes are prefixed by the down (up) arrow ↓ (↑). The attribution rules are written as Haskell-like expressions.

```
Its, It   <↑ uses : [Name] >
Its     = NilIts
          Its.uses = []
        | ConsIts It  Its
          Its1.uses = It.uses ++ Its2.uses
It      → Decl      Name
          It.uses = [Name]
        | Use       Name
          It.uses = [Name]
        | Block     Its
          It.uses = Its.uses

It → Block Its
   Its.env = [project] Its.uses It.env

project :: [Name] → Env → Env
project [] _           = []
project (n:us) env = (getentries n env)++
                     (project us env)
```

Fragment 1: The projection of the environment for the inner blocks.

The semantic function [*project*] projects the environment of the outer most block on the list of identifiers synthesized for each inner block. The semantic function *getentries* is a primitive function defined on environments: given an identifier (*i.e.*, a key), it returns the entries associated with that identifier. These two inductive functions can be efficiently defined within the higher-order attribute grammar formalism. Next, we modelled both functions as a higher-order attribute using the technique of accumulating parameters. Attribute equations are given for nonterminals *Env* and *Uses* which replace the *getentries* and *project* functions. Two higher-order attributes are defined: *getentries* and *project*.

```
Env   <↓ envi : Env, ↑ envo : Env >
Env = NilEnv
      Env1.envo = Env1.envi
    | CEnv   Name  Env
      Env2.envi = if (Name == Env1.id)
                  then CEnv Name Env1.envi
                  else Env1.envi
      Env1.envo = Env2.envo
It   → Block        Its
      ata  project : Uses
      project       = Its.uses
      project.penvi = NilEnv
      project.env   = Its.dclo
      [Its.env]     = [project.penvo]

Uses    <↓ env : Env, ↓ penvi : Env
         , ↑ penvo : Env >
Uses = NilUses
        Uses1.penvo = Uses1.penvi
      | ConsUses Name Uses
        ata  getentries : Env
        getentries       = Uses1.env
        getentries.id    = Name
        getentries.lev   = 0
        getentries.envi  = Uses1.penvi
        Uses2.penvi      = getentries.envo
        Uses2.env        = Uses1.env
        Uses1.penvo      = Uses2.penvo
```

Fragment 2: The environment projection modelled as higher-order attributes.

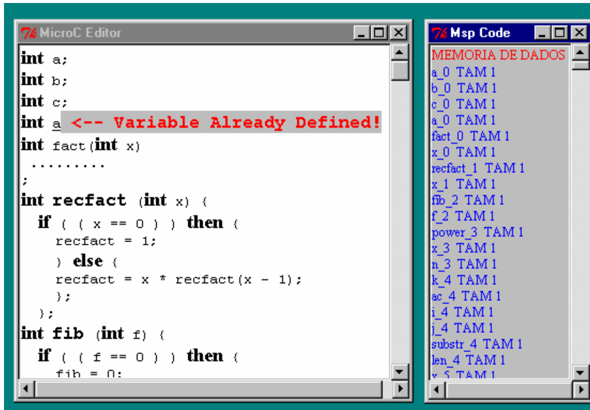

Fig. 2. A language-based environment for MICROC.

Observe that in the higher-order variant we inherit the advantages of the HAG formalism: we are no longer restricted to the implicit positional argument style that is enforced by the conventional functional languages (see the equations of production BLOCK of the previous two fragments). One key aspect of modeling inductive functions as higher-order attributes is the fact that their evaluation becomes automatically incremental. Furthermore, as a result of the ordering computed by standard AG techniques we also guarantee that the underlying inductive computation terminates, which is not ensured when functions are used.

5 Performance Results

The visit-function memoization is the model of incremental evaluation used by the LRC system [KS98]: a generator for incremental language-oriented tools. It produces purely functional C and HASKELL based attribute evaluators using the techniques described in this paper. It generates also lazy circular attribute evaluators and deforested attribute evaluators [SS99].

To benchmark the memoization scheme we consider the C based attribute evaluators and the (strict) incremental engine included in LRC. As input for the beanchmark, we use a traditional language-based editor and a block-structured language, *the* MICROC *language*. The MICROC language is a tiny C based programming language. Figure 2 shows the language-based environment of MICROC constructed with LRC.

The binding and the visit-tree based attribute evaluators generated by LRC for the MICROC HAG perform two traversals over the syntax tree. Two traversals are induced by the AG scheduling algorithm because a *use-before-declare* discipline is allowed in MICROC: declaration of global variables and definition of functions are not required to occur before their first use. Thus, a first traver-

sal collects the environment and a second traversal uses such an environment to detect invalid uses of identifiers. The binding-tree approach uses a single binding-tree to glue together the two traversal functions, while the visit-tree approach uses a visit-tree. Both evaluators construct the same syntax tree in their first traversal. It is worthwhile to note that under a two traversal scheme the binding-tree and the visit-tree approach induce the same number of intermediate trees: a single binding or visit-tree. The overhead due to construct/memoing a large number of intermediate data structures used by the binding-tree approach is not reflected in such evaluators and, thus, in the next results.

Next, we present results obtained when executing the binding-tree and the visit-tree attribute evaluators. We present the number of *cache misses* (functions evaluated), *cache hits* (functions reused), the number of equality tests performed between (shared) terms and the execution time in seconds. We have clocked the execution time on a plain Silicon Graphics workstation.

Model of Evaluation	Attribute Evaluator	Hash Array	Cache Misses	Cache Hits	Equality Tests	Runtime (sec.)
Exhaustive Evaluation	Binding-tree	-	-	-	-	0.23
	Visit-tree	-	-	-	-	0.21
Incremental Evaluation	Binding-tree	10007	4323	1626	9373	0.34
	Visit-tree	10007	3984	910	7232	0.30

The above table presents results obtained both with exhaustive evaluation, *i.e.*, without memoization of the calls to the visit-functions, and with incremental evaluation, *i.e.*, with memoization of the visit-function calls. The results of exhaustive evaluation are very similar. As we explained above, under a two traversal the binding-tree approach does not induce a large of intermediate binding-trees. It should be noticed, however, that the visit-functions that perform the second traversal of the binding-tree evaluator get an additional argument: the binding-tree gluing the traversal functions. As a result, this incremental evaluator has to compare more argument values than the visit-tree based evaluator (see the difference in the number of equality tests). Furthermore, the visit-tree induces fewer misses than the binding-tree approach. That is, fewer visit-functions have to be computed (8% fewer functions are computed). Note that the visit-trees are being specialized for each individual traversal of the evaluator and, thus, they are more likely to share subtrees and, consequently, to share their decorations.

Edit actions: To profile the incremental behaviour of our attribute evaluators we have considered two kinds of modification to the input MICROC: we consider a modification local to the body of a function and a modification that is global to the whole program. To be more precise, we modified the program in two ways: by adding a statement to a function and by adding a global variable. Next, we present the result of the incremental evaluation after the four modifications.

Edit Action	Binding-Tree				Visit-Tree			
	Misses	Hits	Tests	Time	Misses	Hits	Tests	Time
Add a statement	434	432	14376	0.03	434	429	11413	0.03
Add a global variable	3874	1949	165674	0.26	3752	1034	138507	0.24

As expected, the functional incremental evaluators handle local changes extremely well. The execution time in both evaluators is almost negligible. On the contrary, adding a global variable gives poor incremental behaviour. No gain is obtained with the incremental evaluation since almost the same visit-functions are being computed. The exhaustive evaluator achieves a better execution time, since it is not influenced by the interpretative overhead due to the memoization scheme. The incremental behaviour of the microC environment can be greatly improved if we consider the grammar transformation presented in Section 4. Thus, we have transformed the AG in order to project the attribute that defines the environment passed to the body of the microC functions. The environment is projected on the list of identifiers used by the functions.

Edit Action	Binding-Tree				Visit-Tree			
	Misses	Hits	Tests	Time	Misses	Hits	Tests	Time
Add a statement	443	441	19445	0.03	432	431	16175	0.03
Add a global variable	644	813	31444	0.11	522	509	16993	0.10

The projection of the environment drastically improved the performance of both incremental evaluators after a global modification in the input. The number of cache misses and equality tests decreased considerably. As a result, using the projection of attributes, the incremental evaluator gives a speed-up of 2 for global changes when compared to the exhaustive evaluator.

These results show that, even for a simple two traversal evaluator the visit-tree approach presents a better incremental performance. For evaluators that require a large number of traversals, we expect greater differences in performance. For example, the attribute evaluator derived from the Lrc AG (Lrc is a bootstrap system) performs eleven traversals over the tree. Under the binding-tree approach a single nonterminal induces thirty four binding-tree data types. Such binding-trees induce additional arguments to the visit-functions: for example, the visit-function that performs the last visit to that nonterminal symbol gets nine additional arguments. Recall that such arguments have to be tested for equality during incremental evaluation. Under the visit-tree approach, although eleven visit-tree data types still have to be defined, the visit-functions do not get any additional arguments. The Lrc bootstrap grammar also shows how difficult it can be to hand-write a strict, functional attribute evaluator: the writer has to concern himself with partitioning the evaluator into different traversals and to glue the traversal functions.

6 Conclusions

This paper presented two techniques to improve the incremental behaviour of attribute grammars and their functional implementations. Firstly, we defined a new strict, functional implementation of attribute grammars, where a visit-tree is used to convey information between different traversal functions. This technique was implemented in a purely functional attribute grammar. The first experimental results show that our new approach improves the incremental performance of the evaluators by increasing the number of function calls reused, and by decreasing the interpretative overhead of the memoization scheme, when compared to previous functional implementations of AGs.

Secondly, we have defined a transformation for attribute grammars that improves their potential incremental behaviour after a global tree transformation, for all models of incremental evaluation. Our first results show that it drastically increases the performance of the AG. In our experiments we have transformed ourselves the AG. However, an attribute grammar system should apply such transformation automatically.

References

AG93. Andrew W. Appel and Marcelo J. R. Gonçalves. Hash-consing Garbage Collection. Technical Report CS-TR-412-93, Princeton University, Dept. of Computer Science, February 1993. 285

CP96. Alan Carle and Lori Pollock. On the Optimality of Change Propagation for Incremental Evaluation of Hierarchical Attribute Grammars. *ACM Transactions on Programming Languages and Systems*, 18(1):16–29, January 1996. 282

Kas80. Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13:229–256, 1980. 280, 281, 283

KS98. Matthijs Kuiper and João Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In Kay Koskimies, editor, *7th International Conference on Compiler Construction*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, April 1998. 281, 290

M.P94. Jones M.P. The implementation of the gofer functional programming system. Technical Report Research Report YALEU/DCS/RR-1030, Yale University, Dept. of Computer Science, May 1994. 284

Pen94. Maarten Pennings. *Generating Incremental Evaluators*. PhD thesis, Depart. of Computer Science, Utrecht University, The Netherlands, November 1994. 281, 282, 283, 288

PT89. William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *16th Annual ACM Symposium on Principles of Programming Languages*, volume 1, pages 315–328. ACM, January 1989. 288

RT89. T. Reps and T. Teitelbaum. *The Synthesizer Generator*. Springer, 1989. 281

RTD83. Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983. 281

Sar99. João Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, December 1999. 281, 286

SKS97. João Saraiva, Matthijs Kuiper, and Doaitse Swierstra. Effective Function Cache Management for Incremental Attribute Evaluation. In Chris Clark, Tony Davie, and Kevin Hammond, editors, *9th Workshop on Implementation of Functional Languages*, pages 517–528, Scotland, September 1997. 288

SS99. João Saraiva and Doaitse Swierstra. Data Structure Free Compilation. In Stefan Jähnichen, editor, *8th International Conference on Compiler Construction*, volume 1575 of *LNCS*, pages 1–16. Springer-Verlag, March 1999. 290

TC90. Tim Teitelbaum and Richard Chapman. Higher-order attribute grammars and editing environments. In *ACM SIGPLAN'90 Conference on Principles of Programming Languages*, volume 25, pages 197–208. ACM, June 1990. 281, 282

vD92. Leen van Dalen. Incremental evaluation through memoization. Master's thesis, Department of Computer Science, Utrecht University, The Netherlands, August 1992. 284

VSK89. Harald Vogt, Doaitse Swierstra, and Matthijs Kuiper. Higher order attribute grammars. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 131–145. ACM, July 1989. 281