Higher order attribute grammars: a merge between functional and object oriented programming

S.D. Swierstra, H.H. Vogt

RUU-CS-90-12
March 1990

Utrecht University

Department of Computer Science

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

ISSN:0924-3275

Higher Order Attribute Grammars: a Merge between Functional and Object Oriented Programming

S.D. Swierstra, H.H. Vogt

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

March 30, 1990

Abstract

Using incrementally evaluated attribute grammars as a programming language entails the advantanges of both the functional programming style and the object oriented programming style. On the one hand there is a complete absence of the need to explicitely schedule computations in order to maintain functional dependencies between data, whereas on the other hand the underlying syntax trees being edited, capture the concept of a state. In this paper we identify the underlying principles of this dual view and propose extensions to the standard attribute grammar formalisms. A delegation mechanism is introduced in order to deal with the user interface management part of incrementally evaluated attribute grammars. Finally we discuss the use of structure sharing in the efficient implementation of the proposed extensions.

1 Introduction

In this paper we show that attribute grammars are in principle widely applicable, offering a uniform solution to many problems in the area of purely functional systems, in the area of object oriented systems and in the systematic construction of user interfaces.

In the first section we discuss the relationship between the functional and object oriented programming paradigms and attribute grammars. In further sections we describe the extension of the conventional attribute grammar formalism to higher order attribute grammars. This extension makes the formalism more widely applicable. Next we introduce a delegation mechanism, which will be instrumental in the description of user interfaces. In the last section we will give an indication of how these extensions might be implemented at reasonable costs.

Attribute grammars were originally introduced to describe the transduction of programming languages into machine code[Knu68, Knu71]. As such they may, together with affix-grammars and logical programming languages like Prolog, be considered as machine implementable approximations to so-called two-level grammars, which were succesfully used in the description of Algol-68[Wea75, CU77].

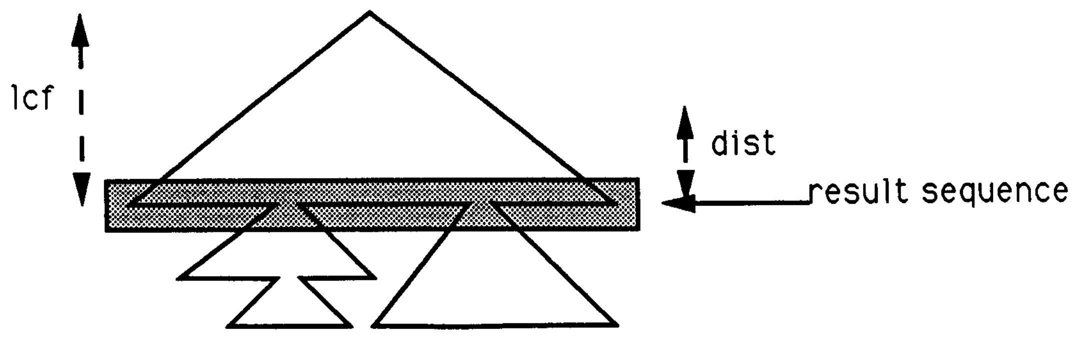


Figure 1:

In the past two decades the emphasis on research on attribute grammars has been on the automatic construction of compilers, of which the efficiency approximates that of hand-written ones. Because most widely used programming languages however were designed with the construction of an efficient hand written compiler in mind, it has in practice proved especially hard to attain similar efficiencies with automatically constructed compilers. This has been the main cause that a lot of actual compiler writers still view attribute grammars as a tool of mainly theoretical relevance, whereas most programmers consider attribute grammars mainly as a compiler construction tool.

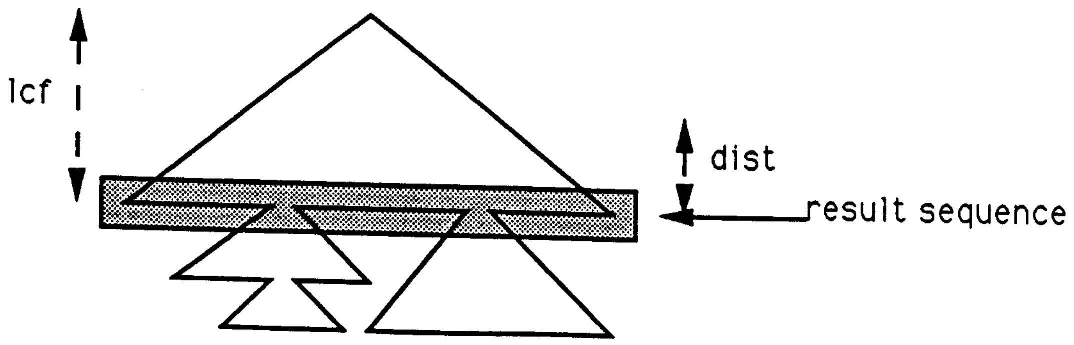
In recent years incrementally evaluated attribute grammars have become increasingly popular[RTD83, RT89]. In these systems the user of the system performs editing operations on an underlying abstract syntax tree and the system automatically maintains the attributes corresponding to the nodes of these trees and the functional dependencies between them. Being originally introduced as a means for the automatic construction of language based editors out of language specifications based on attribute grammars, these systems have appeared to be much wider appliccable[vE88, VBF90, KBHG+87]. Currently systems containing a lot of data from different types, and with complicated relationships between these data, are routinely constructed using e.g. the Synthesizer Generator[RT89].

2 Relation to Other Formalisms

2.1 Attribute Grammars from a Functional Programming View

One of the main advantages of the use of attribute grammars is the static (or equational) character of the specification. The description of relations between data is purely functional, and thus completely void of any sequencing of computations and of explicit garbage collection (i.e. use of assignments). We demonstrate this by giving two formulations for the same problem: take a labelled binary tree and compute the front of the largest complete subtree which shares its top with the original tree. A sketch of the computation and the rôle of some of the identifiers is given in figure 1

The correspondence with functional programming languages is demonstrated by the grammar in figure 2, which has been transcribed into a Miranda[Tur85] program in figure 3. To make the similarity even more striking we have introduced a compact notation for attribute grammars, showing the nonterminals, their functionality (i.e. the set of inherited an synthesized attributes), the various productions and the semantic functions in one single formula.



```
   LTREE(int dist→int* front, int lcf)
(1)::=LTREE, integer, LTREE
        LTREE₁.dist  := LTREE₂.dist := LTREE₀.dist-1
        LTREE₀.lcf   := min(LTREE₁.lcf, LTREE₂.lcf)+1
        LTREE₀.front := if LTREE₁.dist=1→ [integer.val]
                        [] LTREE₁.dist>1→ LTREE₁.front ++ LTREE₂.front
                        fi
(2)|  EMPTY
        LTREE.lcf := 0
        LTREE.front := undefined

   ROOT(→int* front)
(3)::=LTREE
        LTREE.dist := LTREE.lcf
        ROOT.front := LTREE.front
```

Figure 2: Attribute Grammar

```
ltree ::= NODE ltree int ltree | EMPTY
ROOT ::= ltree

eval_ltree(NODE left i right) dist    = (min(leftlcf, rightlcf)+1, front)
        where  (leftlcf, leftfront)   = eval_ltree left (dist-1)
               (rightlcf, rightfront)= eval_ltree rigth (dist-1)
               front                  = dist=1→ [i]
                                        dist>1→ leftfront++rightfront

eval_ltree EMPTY dist = (0, undefined)

ROOT v = front
          where (front, lcf) = eval v lcf
```

Figure 3: Miranda Program

In the program texts `lcf` stands for *level of complete front*, and `dist` is used to locate the level of the complete front in the original tree. Note that inherited attributes in the attribute grammar correspond directly to parameters and synthesized attributes correspond to a field in the result of `eval`. The underlying tree on which the computation is performed is implicit in the attribute grammar description and explicitely present in the Miranda description. The lazy evaluation of the Miranda program allows the use of so-called circular definitions, roughly corresponding to multiple visits in attribute grammars.

Having a single set of synthesized attributes is in direct correspondence with the result of a program transformation called *tupling*. In [KS86] it is shown that this correspondence can be used in transforming functional programs into more efficient ones, thus avoiding the use of e.g. *memo-functions*[Hug85]. Often inherited attributes dependencies are threaded through an abstract syntax tree, which corresponds closely to another functional programming optimisation called *accumulations*[BW88, Bir84].

As a consequence the result of many program transformations which are performed on functional programs in order to increase efficiency, are automatically achieved when using attribute grammars as the starting formalism. This is mainly caused by the fact that in attribute grammars the underlying data structures play a more central role than the associated attributes and functions, whereas in the functional programming case the emphasis is reversed.

From this correspondence it follows that attribute grammars may basically be considered as a functional programming language, without however providing the advantages of many such languages as higher order functions and polymorphism.

2.2 AG's from an Object Oriented View

With the advent of incrementally evaluated attribute grammars the concept of state has entered the arena. Basically the state can be split into two parts:

- *initial part*
  the abstract syntax tree and the *initial attributes*, representing that part of the state which can be manipulated from outside of the system
- *derived part*
  the rest of the attributes, which describe an extension of the state which is (either directly or indirectly) functionally dependent on the initial part of the state

The term *initial attribute* may need some further explanation. In most attribute grammar systems it is assumed that the root of the abstract syntax tree has its inherited attributes filled in by the rest of the system, whereas the synthesized attributes of the terminal symbols are being provided by the scanner. Because we will need a more general class of attributes, i.e. those attributes which do not depend on other attributes, we will introduce so-called *initial attributes.*

When comparing an attribute grammar with an object oriented system we may note the following correspondencies:

grammar	object oriented program
individual nodes	set of objects
tree structure	references between objects
tree transformations	outside messages to objects
attribute updating	inter-object messages

The main difference with most object oriented systems however is that propagating updating information is done implicitely by the system as e.g. in the Higgins[HK88] system, and not explicitely, as in e.g. the Andrew[ea86] system or Smalltalk.

The advantage of this implicit approach is that the extra code associated with correctly scheduling the updating process has not to be provided. Because in object oriented systems this part of the code is extremely hard to get both correct and efficient, this is considered a great advantage.

In conventional object oriented systems there are basically two ways which may be used in maintaining functional dependencies:

- maintaining *view relations*
  In this case an object notifies its so-called *observers* that its value has been changed, and leaves it up to some scheduling mechanism to initiate the updating of those observers. Because of the absence of a formal description of the dependencies underlying a specific system, such a scheduler has to be of a fairly general nature: either the observation relations have to be restricted to a fairly simple form, e.g. simple hierarchies, or potentially very inefficient scheduling has to be accepted.

- sending *difference messages*
  In this case an object sends updating messages to objects depending on it. Thus not only an object has to explicitely maintain which other objects depend on it, but it can also be gleaned from the code on which parts another object depends. Furthermore but it has also to be known *in which way* that other object depends on it. A major disadvantage of this approach is thus that whenever a new object class B is introduced, depending on objects of class A, also the code of A has to be updated.

  An advantage from this approach is that by introducing a large set of messages it can be precisely indicated which arguments of which functional dependencies have changed in which way, and probably costly complete reëvaluations may be avoided. Although this fact is not often noticed, such systems contain a condiderable amount of user programmed finite differencing [PK82] or strength reduction. As a consequence these systems are sometimes hard to understand and maintain.

In the sequel we will show how we may achieve some of the efficiencies of hard-hand-coded object oriented programs, without having to provide all the scheduling details and with maintaining readability, by using an extended attribute grammar formalism.

2.3 Higher Order Attribute Grammars

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 abstract syntax tree would be the optimal form for performing the rest of the computations. Some attempts have been made to alleviate this problem.

Attribute coupled grammars[gG84] allow multi-pass compilers to be elegantly described: the result of an attribute evaluation can be an abstract syntax tree again, which is then used as a starting point for the next pass. In a limited sense this solution has also been chosen in the Synthesizer Generator, where a special attribute computed from the parse tree as delivered by the parser of the input, is taken as the initial tree to perform the attribute computations upon. Furthermore the way in which the attribute representing the unparsed tree is computed may be considered as another attribute coupled grammar.

As a further, probably more esthetical than factual, shortcoming of attribute grammars is that there is usually no correspondence between the grammar part of the system and the functional language which is used to decribe the semantic functions. A direct consequence of this dual-formalism approach is that a lot of properties present in one of the two formalisms is totally absent in the other one, 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 function which are several pages long, and which are directly describable by an attribute grammar.
- in the case of an incrementally evaluated system the semantic functions do not profit from this incrementality property, and are, in the case of re-evaluation completely re-evaluated.

Higher order attribute grammars[VSK89] 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.

In the previous section we have seen that partially parametrising an evaluation function with a tree results in a function mapping inherited to synthesized attributes. Because of this close correspondence between trees and functions the term *higher-order* was coined for grammars allowing this kind of attributes.

In [VSK89] an example is given in which a multi-pass compiler is being described using higher order attributes. Here we will demonstrate another use of such attributes: the possibility to avoid the use of separate semantic functions. In figure 4 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 containg the index positions of the applied occurences 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 example the following can be noted:

- The attribute *env* is a higher order attribute. The tree structure is built using the *constructor functions* ENV_6 and ENV_7, which correspond to the respective productions for *ENV*. The attribute *env* is instantiated (i.e. a copy of the tree is attributed) in the occurences of the first production of *APPS*, and takes the rôle of a semantic function.

```
ROOT( → int* seq)
(1)::= let DECLS in APPS ni
        APPS.env := DECLS.env
        ROOT.seq := APPS.seq

DECLS( → int number, ENV env)
(2)::= DECLS, identifier
        DECLS_0.number := DECLS_1.number
        DECLS_0.env := ENV_6([identifier.id, DECLS_1.number](DECLS_1.env))
(3)|   EMPTY
        DECLS.env := ENV_7
        DECLS.number := 1

APPS(ENV env → int* seq)
(4)::= APPS, identifier, env
        APPS_0.seq := APPS_1.seq ++ [env.index]
        env.param := identifier.id
(5)|   EMPTY
        APPS.seq := []

ENV(ID param → int index)
(6)::= [ID id, int number], ENV
        ENV_0.index := if ENV_0.param=id → number
                       [] ENV_0.param≠id → ENV_1.index
                       fi
        ENV_1.param := ENV_0.param
(7)|   EMPTY
       ENV.index := errorvalue
```

Figure 4: A Higher Order Attribute Grammar

- The first alternative of the nonterminal *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.
- 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 the last section this brings many consequences for the implementation, and some opportunities for optimisation.

We finish this section by noticing that this grammar too may be directly transcribed into a functional language. For the representation of the higher order attribute we have two choices:

1. either we construct a recursive data structure, and at the place of the instatiation we place a call of a function *eval_env* to which this data structure and the inherited attributes are passed
2. or we partially parameterize a function *eval_env'* with the two initial attributes and the function representing its descendant, and use this function at the place of the instantiation.

3 Handling User Interaction and Message Passing

In most incrementally evaluated attribute grammar systems the handling of user interaction is not described using the formalism itself. In the Synthesizer Generator the unparsing of a tree closely reflects the structure of the abstract syntax tree, and for good reasons. Because there is no general way for describing which characters on the screen refer to which nodes in the tree a simple, invertable unparsing scheme had to be chosen.

The problem that one needs more flexibility in unparsing than merely giving some flattened form of the tree has been widely recognised[HT86] and solving this problem in a wider sense is one of the main research topics in the area of user interface management systems. Considering language based editing it might be desirable e.g. to have separate windows representing:

- the nesting structure of the procedures
- the text of a specific procedure
- the text documenting that procedure
- a number of program fragments containing calls to that procedure

In such a system one might imagine that the contents of the last three windows is automatically updated, when a specific procedure is selected in the first window.

In [FZ88] a two-dimensional unparsing scheme is introduced, using a preprocessor which adds a number of hidden attributes to the tree used to represent the position of the boxes on the screen which correspond to the nodes in the tree. However the hierarchical structure of the parse tree still dictates the hierarchical structure of the unparsed representation.

It is here that higher order attribute grammars may play an important rôle in describing the *user interface*. Many alternative representations of the initial structure may be computed in higher order attributes. These attributes may be passed through the tree and combined in several ways. Finaly these structures will be unparsed into a representation on the screen. Although this may seem to be quite a step away from normal attribute computation, actually all most conventional application programs do is computing new data out of existing data, and using this data as a further basis for computation. The main difference in our approach is that this is done in a functional style and in an incremental way.

We make the following assumptions:

- external representations are closely associated with a higher order attribute tree in the application. Pointing at a screen is easily mapped onto selecting a node in that tree. We will call the nodes of such a tree *visible nodes*. Most user interface management systems (e.g. [SG86]) will provide some form of such an inverse mapping.
- for every nonterminal in the grammar a number of *transformations* is defined. When a transformation is applied to a specific node such a transformation either changes the tree at that position, provided the node is part of the initial tree, and/or changes the value of an initial attribute.

As we have seen the user of the system is presented some external representation, computed from internal data; a computation which may have gone through several steps before resulting in the displayed data.

The problem to be solved now (the *pointing problem*) is that visible nodes may be selected, whereas initial nodes have to be transformed. As a consequence a mapping from visible to initial nodes has to be realised, which is achieved as follows:

- for every derived node (and thus for visible nodes), it is maintained at which node it was constructed. This relation may be used to trace back for each node, and especially the visible nodes, the initial node from which it finally originated. The nodes on the chain to that initial node we will call its *ancestors*.
- when a visible node is selected, all its transformations, and the transformations of its ancestors, are made selectable.
- when a specific transformation is selected, the transformation is performed, and the usual update mechanism is started. This will most likely have as a result that the final external representation is updated, thus giving the user a visible feedback of the change to the initial data.

In the object oriented terminology one would say that when a node cannot handle a specific transformation, this transformation is *delegated* to its ancestor. In figure 5 a chain of nodes is depicted, together with a set of possible transformations at each node. When the indicated sub-structure of the external representation is selected, all transformations associated with nodes A, B and C become available and may be selected.

At first sight it may seem undesirable to allow transformations on non-initial nodes, because this would break-up the functional dependencies. We allow however transformations which change initial attributes of intermediate ancestors for the following reason.

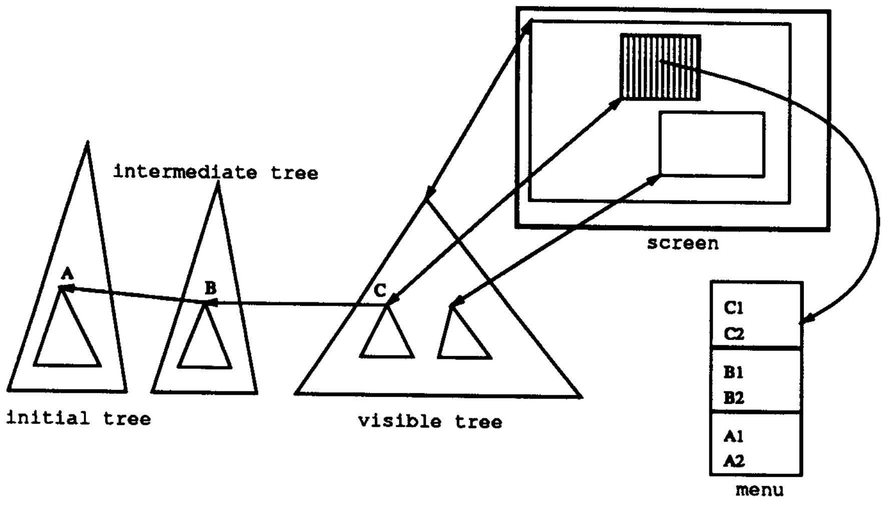


Figure 5: A Node Chain with Transformations

When constructing some form of external representation, it is often not clear how to choose certain aspects of this external representation. Examples of this are its size or the location on the screen, or what part of the total data structure should be represented by visible nodes. Default choices can be made by providing specific values for the initial attributes. It is however essential to be able to change the value of these initial attributes, without having to change the initial data.

We finish this section by noting that there is no straightforward way to map this mechanism onto a purely functional language. This is amongst others caused by the fact that we now may have different states corresponding to the same initial state.

4 Incremental Evaluation of Higher Order Attribute Grammars

In this section we discuss how structure sharing is used for the efficient implementation of incrementally evaluated higher order attribute grammars.

Various strategies for incremental evaluation of attribute grammars have been developed [Rep82, Rep84, Yeh83, Hoo86]. It is no surprise that any extension to the basic attribute grammar formalism comes with its associated increase in complexity of the evaluation strategy [RMT86]. Further complicating factors are thus to be expected from introducing higher order attributes.

We note the following complicating aspects:

- Because a higher order attribute may be instantiated at several positions, as e.g. the *env* attribute in the example, several sets of attributes may be associated with a single syntax tree. This is comparable with the full functional programming case were a procedure is in general invoked more than once. It is a considerable extension of the conventional attribute grammar case, in which every tree occurs only once, and the incremental updating process is using the original tree as a simple cache structure.

- It may be the case that during attribute evaluation the structure of the total tree is changed because new values get assigned to higher order attributes. As a consequence some of the attributes which may be assumed by the updating algorithm to be affected, may appear to have become inaccessible during the updating process.
- There is ample opportunity for structure sharing[Chr89], because higher order attributes are inherently structural objects.
- As demonstrated by the example it may be the case that there are several instantiations of a higher order attribute in existence which have the same inherited attributes. This is caused by the fact that we use attributes to model semantic functions. It is desirable to indentify these identical expansions in order to avoid superfluous computations. When an environment has changed it is desirable that the consequences of this updating are computed for every distinct identifier, but not for every identifier occurrence.

In [VSK89] a transformation is given, which maps a higher order grammar onto a conventional one. The basic step in this mapping is the addition of extra dependencies to productions in which higher order attributes are instantiated. These dependencies are used to indicate that the attributes of the instantiation depend on the higher order attribute, which should thus be evaluated before visiting the tree described by it. All grammar based dependency analysis may be done on the transformed grammar.

In order to get reasonable efficiency we expect to make extensive use of structure sharing. In order to do so we maintain two heaps, the ***structure heap SH*** and the ***instantiation heap IH***. Higher order attributes are represented by pointers into *SH*, and the evaluation of every node-constructing semantic function results in a new structure in *SH*. From a functional programming point of view we may consider every pointer into *SH* as uniquely identifying an evaluation function, partially parameterised with an abstract syntax tree.

Nodes in the attributed tree are represented by four-tuples in *IH*:

1. A pointer *S* into *SH*, representing the structure of the attributed tree of which this node is the top.
2. The values of the inherited attributes *IV*.
3. The values of the synthesized attributes *SV*.
4. Pointers *SP* to the nodes of its sons. These pointers may be **nil** when the attributes of the sons have not yet been evaluated.

Finally we maintain a partial mapping *M*:

$$\text{(structure-pointer} \times \text{inherited attributes)} \rightarrow \text{node-pointer.}$$

We will now first consider the simple case of a nonterminal which is only visited once during an evaluation. When the node with structure *s* gets scheduled for evaluation the following steps are taken:

1. evaluate the values *ih* of the inherited attributes
2. if $s \times ih$ is in the domain of *M*, then share the attributed tree *M(s, ih)*.
3. if not create a node in *IH*, containing the values *s* and *ih* and visit that node.

In this way we achieve a *memo-function* implementation[Hug85]. Note that in this way all instantiations of a higher order attribute, when called with the same arguments, are shared and evaluated only once. An interesting aspect is now the following observation. When, as a result of reevaluation, a value in the *SH* changes, the nodes in *IH* referring to it will be marked for reevaluation. Because there are only as many nodes existent as there are different sets of inherited attributes a considerable increase in speed may be achieved. Not all instantiations, but only all different instantiations have to be reevaluated. Furthermore, those subtrees for which the reevaluation results in the same synthesized attributes as before, may be left out from the data structure maintained for rescheduling.

We now consider the case in which multiple visits may be necessary, and assume that the visits are ordered in some way. The solution is now to consider each visit as not only returning its associated synthesized attributes, but also the partially parameterised function corresponding to the as yet unevaluated part of the tree. This function is readily identified with the node in the *IH*, containing the inherited and synthesized attributes of the visit. So by extending the domain of the function *M* to $(SH \cup EH \times IH)$ and making the first component of the structures in the *IH* not only refer to elements in *SH* but also to elements in *IH*, the aforementioned memo-isation can be extended to the multiple visit case.

5 Conclusions

The correspondence of attribute grammars with functional programs has been indicated. We have shown how by using higher order attribute grammars, coupled with chaining nodes in the trees in order to find the appropriate places for performing the indicated transformations, the application area of attribute grammars can be considerably extended.

We expect that by further pursuing this similarity, and introducing some of the elegance which functional programs exhibit into the attribute arena, attribute grammars can be turned into a widely used programming formalism.

References

[Bir84] Richard S. Bird¡. The promotion and accumulation strategies in transformational programming. *TOPLAS*, 6(4):487–504, 1984.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* International Series in Computer Science. Prentice Hall, 1988.

[Chr89] Henning Christiansan. Structure sharing in incremental systems. *Structured Programming*, 10(4):169–186, 1989.

[CU77] J. Craig Cleaveland and Robert C. Uzgalis. *Grammars for Programming Languages.* Elsevier Computer Science Library, 1977.

[FZ88] Paul Franchi-Zannettacci. Attribute specifications for graphical interface generation. Rapport de Recherche 937, INRIA, Décembre 1988.

[GG84] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *SIGPLAN Notices*, pages 157–170. ACM, 1984.

[HK88] Scott E. Hudson and Roger King. Semantic feedback in the higgens uims. *IEEE Transactions on Software Engineering*, 14(8):1188–1206, August 1988.

[Hoo86] Roger Scott Hoover. Dynamically bypassing copy rule chains in attribute grammars. In *Proceedings 13th Annual ACM Symposium on Principles of Programming Languages*, pages 14–25, 1986.

[HT86] Susan Horwitz and Tim Teitelbaum. Generating editing environments based on relations and attributes. *TOPLAS*, 8(4):577–608, October 1986.

[Hug85] J. Hughes. Lazy memo-functions. In J-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 129–146. Springer, 1985.

[KBHG+87] B. Krieg-Brückner, B. Hoffmann, H. Ganzinger, M. Broy, R. Wilhelm, U. Möncke, B. Weisberger, A. McGettrick, I.G. Cambell, and G. Winterstein. Program development by specification and transformation. In *Proceedings of ESPRIT Conference 1986*. North-Holland, 1987.

[Knu68] D.E. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127–145, 1968.

[Knu71] D.E. Knuth. Semantics of context-free languages (correction). *Math. Syst. Theory*, 5(1):95–96, 1971.

[KS86] M.F. Kuiper and S.D. Swierstra. Using attribute grammars to derive efficient functional programs. Technical Report RUU-CS-86-16, Dept. of Computer Science, P.O.Box 80.089, 3508 TB Utrecht, the Netherlands, 1986.

[Mea86] James H. Morris and et al. Andrew: A distributed personal computing environment. *CACM*, 29(3):184–201, March 1986.

[PK82] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *TOPLAS*, 4(3):402–454, 1982.

[Rep82] Thomas Reps. Optimal-time incremental semantic analysis for syntax directed editors. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 169–176, January 1982.

[Rep84] Thomas Reps. *Generating Language Based Environments.* PhD thesis, M.I.T. Press, 1984.

[RMT86] Thomas Reps, Carla Marceau, and Tim Teitelbaum. Remote attribute updating for language based editors. In *13th Annual ACm Conference on the Principles of Programming Languages*, pages 1–13. ACM, 1986.

[RT89] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual.* Springer, third edition, 1989.

[RTD83] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language based editors. *TOPLAS*, 5(3):449–477, July 1983.

[SG86] Robert W. Schleifer and Jim Gettys. The x window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[Tur85] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 1–16. Springer, 1985.

[VBF90] Harald Vogt, Aswin van den Berg, and Arend Freije. Amazingly fast development of a program transformation system with attribute grammars and dynamic transformations. In *this conference??*, 1990.

[vE88] Peter H.J. van Eijk. *Software Tools for the Specification Language Lotos.* PhD thesis, Twente University, P.O.Box 217, 7500 AE Enschede, the Netherlands, 1988.

[VSK89] Harald H. Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher order attribute grammars. In *SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland, Oregon*, pages 131–145. ACM, 1989.

[Wea75] A. van Wijngaarden et. al. *Revised Report on the Algorithmic Language Algol 68.* Springer, 1975.

[Yeh83] Dashing Yeh. On incremental evaluation of ordered attributed grammars. *BIT*, 23:308–320, 1983.

Higher order attribute grammars: a merge between functional and object oriented programming

S.D. Swierstra, H.H. Vogt

RUU-CS-90-12
March 1990

Utrecht University

Department of Computer Science

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

Higher order attribute grammars: a merge between functional and object oriented programming

S.D. Swierstra, H.H. Vogt

Technical Report RUU-CS-90-12
March 1990

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

ISSN:0924-3275

Higher Order Attribute Grammars: a Merge between Functional and Object Oriented Programming

S.D. Swierstra, H.H. Vogt

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

March 30, 1990

Abstract

Using incrementally evaluated attribute grammars as a programming language entails the advantanges of both the functional programming style and the object oriented programming style. On the one hand there is a complete absence of the need to explicitely schedule computations in order to maintain functional dependencies between data, whereas on the other hand the underlying syntax trees being edited, capture the concept of a state. In this paper we identify the underlying principles of this dual view and propose extensions to the standard attribute grammar formalisms. A delegation mechanism is introduced in order to deal with the user interface management part of incrementally evaluated attribute grammars. Finally we discuss the use of structure sharing in the efficient implementation of the proposed extensions.

1 Introduction

In this paper we show that attribute grammars are in principle widely applicable, offering a uniform solution to many problems in the area of purely functional systems, in the area of object oriented systems and in the systematic construction of user interfaces.

In the first section we discuss the relationship between the functional and object oriented programming paradigms and attribute grammars. In further sections we describe the extension of the conventional attribute grammar formalism to higher order attribute grammars. This extension makes the formalism more widely applicable. Next we introduce a delegation mechanism, which will be instrumental in the description of user interfaces. In the last section we will give an indication of how these extensions might be implemented at reasonable costs.

Attribute grammars were originally introduced to describe the transduction of programming languages into machine code[Knu68, Knu71]. As such they may, together with affix-grammars and logical programming languages like Prolog, be considered as machine implementable approximations to so-called two-level grammars, which were succesfully used in the description of Algol-68[Wea75, CU77].

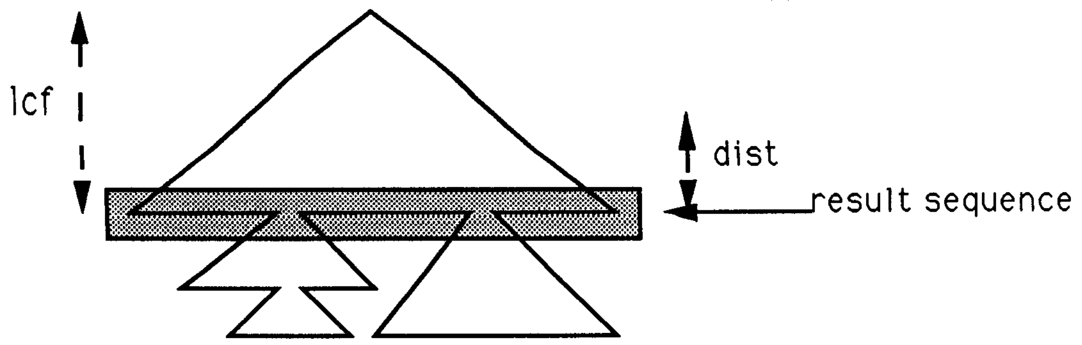


Figure 1:

In the past two decades the emphasis on research on attribute grammars has been on the automatic construction of compilers, of which the efficiency approximates that of hand-written ones. Because most widely used programming languages however were designed with the construction of an efficient hand written compiler in mind, it has in practice proved especially hard to attain similar efficiencies with automatically constructed compilers. This has been the main cause that a lot of actual compiler writers still view attribute grammars as a tool of mainly theoretical relevance, whereas most programmers consider attribute grammars mainly as a compiler construction tool.

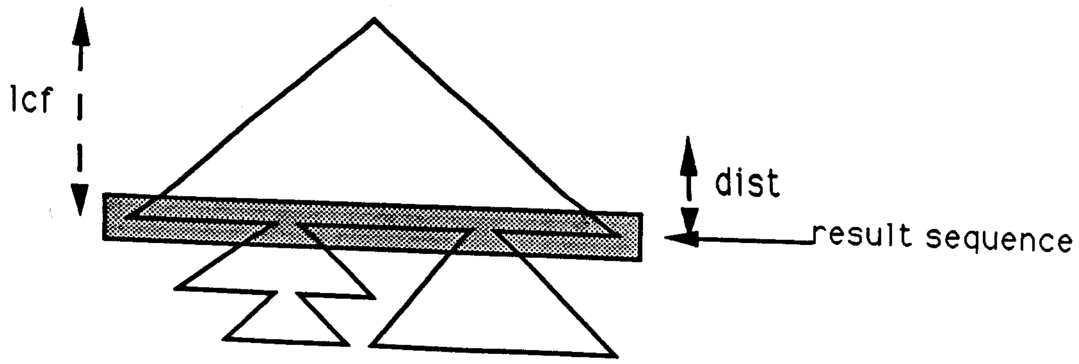
In recent years incrementally evaluated attribute grammars have become increasingly popular[RTD83, RT89]. In these systems the user of the system performs editing operations on an underlying abstract syntax tree and the system automatically maintains the attributes corresponding to the nodes of these trees and the functional dependencies between them. Being originally introduced as a means for the automatic construction of language based editors out of language specifications based on attribute grammars, these systems have appeared to be much wider appliccable[vE88, VBF90, KBHG+87]. Currently systems containing a lot of data from different types, and with complicated relationships between these data, are routinely constructed using e.g. the Synthesizer Generator[RT89].

2 Relation to Other Formalisms

2.1 Attribute Grammars from a Functional Programming View

One of the main advantages of the use of attribute grammars is the static (or equational) character of the specification. The description of relations between data is purely functional, and thus completely void of any sequencing of computations and of explicit garbage collection (i.e. use of assignments). We demonstrate this by giving two formulations for the same problem: take a labelled binary tree and compute the front of the largest complete subtree which shares its top with the original tree. A sketch of the computation and the rôle of some of the identifiers is given in figure 1

The correspondence with functional programming languages is demonstrated by the grammar in figure 2, which has been transcribed into a Miranda[Tur85] program in figure 3. To make the similarity even more striking we have introduced a compact notation for attribute grammars, showing the nonterminals, their functionality (i.e. the set of inherited an synthesized attributes), the various productions and the semantic functions in one single formula.



```
  LTREE(int dist→int* front, int lcf)
(1)::=LTREE, integer, LTREE
        LTREE_1.dist  := LTREE_2.dist := LTREE_0.dist-1
        LTREE_0.lcf   := min(LTREE_1.lcf, LTREE_2.lcf)+1
        LTREE_0.front := if LTREE_1.dist=1→ [integer.val]
                            [] LTREE_1.dist>1→ LTREE_1.front ++ LTREE_2.front
                            fi
(2)|  EMPTY
        LTREE.lcf := 0
        LTREE.front := undefined

  ROOT(→int* front)
(3)::=LTREE
        LTREE.dist := LTREE.lcf
        ROOT.front := LTREE.front
```

Figure 2: Attribute Grammar

```
ltree ::= NODE ltree int ltree | EMPTY
ROOT ::= ltree

eval_ltree(NODE left i right) dist   = (min(leftlcf, rightlcf)+1, front)
        where  (leftlcf, leftfront)   = eval_ltree left (dist-1)
               (rightlcf, rightfront)= eval_ltree rigth (dist-1)
               front                 = dist=1→ [i]
                                       dist>1→ leftfront++rightfront

eval_ltree EMPTY dist = (0, undefined)

ROOT v = front
            where (front, lcf) = eval v lcf
```

Figure 3: Miranda Program

In the program texts `lcf` stands for *level of complete front*, and `dist` is used to locate the level of the complete front in the original tree. Note that inherited attributes in the attribute grammar correspond directly to parameters and synthesized attributes correspond to a field in the result of `eval`. The underlying tree on which the computation is performed is implicit in the attribute grammar description and explicitely present in the Miranda description. The lazy evaluation of the Miranda program allows the use of so-called circular definitions, roughly corresponding to multiple visits in attribute grammars.

Having a single set of synthesized attributes is in direct correspondence with the result of a program transformation called *tupling*. In [KS86] it is shown that this correspondence can be used in transforming functional programs into more efficient ones, thus avoiding the use of e.g. *memo-functions*[Hug85]. Often inherited attributes dependencies are threaded through an abstract syntax tree, which corresponds closely to another functional programming optimisation called *accumulations*[BW88, Bir84].

As a consequence the result of many program transformations which are performed on functional programs in order to increase efficiency, are automatically achieved when using attribute grammars as the starting formalism. This is mainly caused by the fact that in attribute grammars the underlying data structures play a more central role than the associated attributes and functions, whereas in the functional programming case the emphasis is reversed.

From this correspondence it follows that attribute grammars may basically be considered as a functional programming language, without however providing the advantages of many such languages as higher order functions and polymorphism.

2.2 AG's from an Object Oriented View

With the advent of incrementally evaluated attribute grammars the concept of state has entered the arena. Basically the state can be split into two parts:

- *initial part*
  the abstract syntax tree and the *initial attributes*, representing that part of the state which can be manipulated from outside of the system
- *derived part*
  the rest of the attributes, which describe an extension of the state which is (either directly or indirectly) functionally dependent on the initial part of the state

The term *initial attribute* may need some further explanation. In most attribute grammar systems it is assumed that the root of the abstract syntax tree has its inherited attributes filled in by the rest of the system, whereas the synthesized attributes of the terminal symbols are being provided by the scanner. Because we will need a more general class of attributes, i.e. those attributes which do not depend on other attributes, we will introduce so-called *initial attributes.*

When comparing an attribute grammar with an object oriented system we may note the following correspondencies:

grammar	object oriented program
individual nodes	set of objects
tree structure	references between objects
tree transformations	outside messages to objects
attribute updating	inter-object messages

The main difference with most object oriented systems however is that propagating updating information is done implicitely by the system as e.g. in the Higgins[HK88] system, and not explicitely, as in e.g. the Andrew[ea86] system or Smalltalk.

The advantage of this implicit approach is that the extra code associated with correctly scheduling the updating process has not to be provided. Because in object oriented systems this part of the code is extremely hard to get both correct and efficient, this is considered a great advantage.

In conventional object oriented systems there are basically two ways which may be used in maintaining functional dependencies:

- maintaining *view relations*
  In this case an object notifies its so-called *observers* that its value has been changed, and leaves it up to some scheduling mechanism to initiate the updating of those observers. Because of the absence of a formal description of the dependencies underlying a specific system, such a scheduler has to be of a fairly general nature: either the observation relations have to be restricted to a fairly simple form, e.g. simple hierarchies, or potentially very inefficient scheduling has to be accepted.

- sending *difference messages*
  In this case an object sends updating messages to objects depending on it. Thus not only an object has to explicitely maintain which other objects depend on it, but it can also be gleaned from the code on which parts another object depends. Furthermore but it has also to be known *in which way* that other object depends on it. A major disadvantage of this approach is thus that whenever a new object class B is introduced, depending on objects of class A, also the code of A has to be updated.

  An advantage from this approach is that by introducing a large set of messages it can be precisely indicated which arguments of which functional dependencies have changed in which way, and probably costly complete reëvaluations may be avoided. Although this fact is not often noticed, such systems contain a condiderable amount of user programmed finite differencing [PK82] or strength reduction. As a consequence these systems are sometimes hard to understand and maintain.

In the sequel we will show how we may achieve some of the efficiencies of hard-hand-coded object oriented programs, without having to provide all the scheduling details and with maintaining readability, by using an extended attribute grammar formalism.

2.3 Higher Order Attribute Grammars

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 abstract syntax tree would be the optimal form for performing the rest of the computations. Some attempts have been made to alleviate this problem.

Attribute coupled grammars[gG84] allow multi-pass compilers to be elegantly described: the result of an attribute evaluation can be an abstract syntax tree again, which is then used as a starting point for the next pass. In a limited sense this solution has also been chosen in the Synthesizer Generator, where a special attribute computed from the parse tree as delivered by the parser of the input, is taken as the initial tree to perform the attribute computations upon. Furthermore the way in which the attribute representing the unparsed tree is computed may be considered as another attribute coupled grammar.

As a further, probably more esthetical than factual, shortcoming of attribute grammars is that there is usually no correspondence between the grammar part of the system and the functional language which is used to decribe the semantic functions. A direct consequence of this dual-formalism approach is that a lot of properties present in one of the two formalisms is totally absent in the other one, 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 function which are several pages long, and which are directly describable by an attribute grammar.
- in the case of an incrementally evaluated system the semantic functions do not profit from this incrementality property, and are, in the case of re-evaluation completely re-evaluated.

Higher order attribute grammars[VSK89] 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.

In the previous section we have seen that partially parametrising an evaluation function with a tree results in a function mapping inherited to synthesized attributes. Because of this close correspondence between trees and functions the term *higher-order* was coined for grammars allowing this kind of attributes.

In [VSK89] an example is given in which a multi-pass compiler is being described using higher order attributes. Here we will demonstrate another use of such attributes: the possibility to avoid the use of separate semantic functions. In figure 4 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 containg the index positions of the applied occurences 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 example the following can be noted:

- The attribute *env* is a higher order attribute. The tree structure is built using the *constructor functions* ENV_6 and ENV_7, which correspond to the respective productions for *ENV*. The attribute *env* is instantiated (i.e. a copy of the tree is attributed) in the occurences of the first production of *APPS*, and takes the rôle of a semantic function.

```
ROOT( → int* seq)
(1)::= let DECLS in APPS ni
        APPS.env := DECLS.env
        ROOT.seq := APPS.seq

DECLS( → int number, ENV env)
(2)::= DECLS, identifier
        $DECLS_0$.number := $DECLS_1$.number
        $DECLS_0$.env := $ENV_6$([identifier.id, $DECLS_1$.number]($DECLS_1$.env))
(3)|   EMPTY
        DECLS.env := $ENV_7$
        DECLS.number := 1

APPS(ENV env → int* seq)
(4)::= APPS, identifier, env
        $APPS_0$.seq := $APPS_1$.seq ++ [env.index]
        env.param := identifier.id
(5)|   EMPTY
        APPS.seq := []

ENV(ID param → int index)
(6)::= [ID id, int number], ENV
        $ENV_0$.index := if $ENV_0$.param=id → number
                         [] $ENV_0$.param≠id → $ENV_1$.index
                         fi
        $ENV_1$.param := $ENV_0$.param
(7)|   EMPTY
       ENV.index := errorvalue
```

Figure 4: A Higher Order Attribute Grammar

- The first alternative of the nonterminal *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.

- 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 the last section this brings many consequences for the implementation, and some opportunities for optimisation.

We finish this section by noticing that this grammar too may be directly transcribed into a functional language. For the representation of the higher order attribute we have two choices:

1. either we construct a recursive data structure, and at the place of the instatiation we place a call of a function *eval_env* to which this data structure and the inherited attributes are passed

2. or we partially parameterize a function *eval_env'* with the two initial attributes and the function representing its descendant, and use this function at the place of the instantiation.

3 Handling User Interaction and Message Passing

In most incrementally evaluated attribute grammar systems the handling of user interaction is not described using the formalism itself. In the Synthesizer Generator the unparsing of a tree closely reflects the structure of the abstract syntax tree, and for good reasons. Because there is no general way for describing which characters on the screen refer to which nodes in the tree a simple, invertable unparsing scheme had to be chosen.

The problem that one needs more flexibility in unparsing than merely giving some flattened form of the tree has been widely recognised[HT86] and solving this problem in a wider sense is one of the main research topics in the area of user interface management systems. Considering language based editing it might be desirable e.g. to have separate windows representing:

- the nesting structure of the procedures
- the text of a specific procedure
- the text documenting that procedure
- a number of program fragments containing calls to that procedure

In such a system one might imagine that the contents of the last three windows is automatically updated, when a specific procedure is selected in the first window.

In [FZ88] a two-dimensional unparsing scheme is introduced, using a preprocessor which adds a number of hidden attributes to the tree used to represent the position of the boxes on the screen which correspond to the nodes in the tree. However the hierarchical structure of the parse tree still dictates the hierarchical structure of the unparsed representation.

It is here that higher order attribute grammars may play an important rôle in describing the *user interface.* Many alternative representations of the initial structure may be computed in higher order attributes. These attributes may be passed through the tree and combined in several ways. Finaly these structures will be unparsed into a representation on the screen. Although this may seem to be quite a step away from normal attribute computation, actually all most conventional application programs do is computing new data out of existing data, and using this data as a further basis for computation. The main difference in our approach is that this is done in a functional style and in an incremental way.

We make the following assumptions:

- external representations are closely associated with a higher order attribute tree in the application. Pointing at a screen is easily mapped onto selecting a node in that tree. We will call the nodes of such a tree *visible nodes.* Most user interface management systems (e.g. [SG86]) will provide some form of such an inverse mapping.
- for every nonterminal in the grammar a number of *transformations* is defined. When a transformation is applied to a specific node such a transformation either changes the tree at that position, provided the node is part of the initial tree, and/or changes the value of an initial attribute.

As we have seen the user of the system is presented some external representation, computed from internal data; a computation which may have gone through several steps before resulting in the displayed data.

The problem to be solved now (the *pointing problem*) is that visible nodes may be selected, whereas initial nodes have to be transformed. As a consequence a mapping from visible to initial nodes has to be realised, which is achieved as follows:

- for every derived node (and thus for visible nodes), it is maintained at which node it was constructed. This relation may be used to trace back for each node, and especially the visible nodes, the initial node from which it finally originated. The nodes on the chain to that initial node we will call its *ancestors.*
- when a visible node is selected, all its transformations, and the transformations of its ancestors, are made selectable.
- when a specific transformation is selected, the transformation is performed, and the usual update mechanism is started. This will most likely have as a result that the final external representation is updated, thus giving the user a visible feedback of the change to the initial data.

In the object oriented terminology one would say that when a node cannot handle a specific transformation, this transformation is *delegated* to its ancestor. In figure 5 a chain of nodes is depicted, together with a set of possible transformations at each node. When the indicated sub-structure of the external representation is selected, all transformations associated with nodes A, B and C become available and may be selected.

At first sight it may seem undesirable to allow transformations on non-initial nodes, because this would break-up the functional dependencies. We allow however transformations which change initial attributes of intermediate ancestors for the following reason.

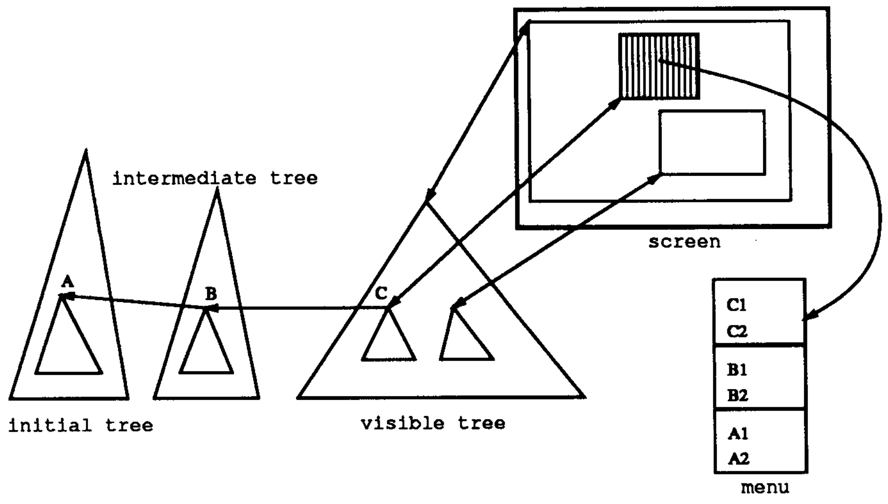


Figure 5: A Node Chain with Transformations

When constructing some form of external representation, it is often not clear how to choose certain aspects of this external representation. Examples of this are its size or the location on the screen, or what part of the total data structure should be represented by visible nodes. Default choices can be made by providing specific values for the initial attributes. It is however essential to be able to change the value of these initial attributes, without having to change the initial data.

We finish this section by noting that there is no straightforward way to map this mechanism onto a purely functional language. This is amongst others caused by the fact that we now may have different states corresponding to the same initial state.

4 Incremental Evaluation of Higher Order Attribute Grammars

In this section we discuss how structure sharing is used for the efficient implementation of incrementally evaluated higher order attribute grammars.

Various strategies for incremental evaluation of attribute grammars have been developed [Rep82, Rep84, Yeh83, Hoo86]. It is no surprise that any extension to the basic attribute grammar formalism comes with its associated increase in complexity of the evaluation strategy [RMT86]. Further complicating factors are thus to be expected from introducing higher order attributes.

We note the following complicating aspects:

- Because a higher order attribute may be instantiated at several positions, as e.g. the *env* attribute in the example, several sets of attributes may be associated with a single syntax tree. This is comparable with the full functional programming case were a procedure is in general invoked more than once. It is a considerable extension of the conventional attribute grammar case, in which every tree occurs only once, and the incremental updating process is using the original tree as a simple cache structure.

- It may be the case that during attribute evaluation the structure of the total tree is changed because new values get assigned to higher order attributes. As a consequence some of the attributes which may be assumed by the updating algorithm to be affected, may appear to have become inaccessible during the updating process.
- There is ample opportunity for structure sharing[Chr89], because higher order attributes are inherently structural objects.
- As demonstrated by the example it may be the case that there are several instantiations of a higher order attribute in existence which have the same inherited attributes. This is caused by the fact that we use attributes to model semantic functions. It is desirable to indentify these identical expansions in order to avoid superfluous computations. When an environment has changed it is desirable that the consequences of this updating are computed for every distinct identifier, but not for every identifier occurrence.

In [VSK89] a transformation is given, which maps a higher order grammar onto a conventional one. The basic step in this mapping is the addition of extra dependencies to productions in which higher order attributes are instantiated. These dependencies are used to indicate that the attributes of the instantiation depend on the higher order attribute, which should thus be evaluated before visiting the tree described by it. All grammar based dependency analysis may be done on the transformed grammar.

In order to get reasonable efficiency we expect to make extensive use of structure sharing. In order to do so we maintain two heaps, the *structure heap SH* and the *instantiation heap IH*. Higher order attributes are represented by pointers into *SH*, and the evaluation of every node-constructing semantic function results in a new structure in *SH*. From a functional programming point of view we may consider every pointer into *SH* as uniquely identifying an evaluation function, partially parameterised with an abstract syntax tree.

Nodes in the attributed tree are represented by four-tuples in *IH*:

1. A pointer *S* into *SH*, representing the structure of the attributed tree of which this node is the top.
2. The values of the inherited attributes *IV*.
3. The values of the synthesized attributes *SV*.
4. Pointers *SP* to the nodes of its sons. These pointers may be **nil** when the attributes of the sons have not yet been evaluated.

Finally we maintain a partial mapping *M*:

$$\text{(structure-pointer}\times\text{inherited attributes)}\rightarrow\text{node-pointer.}$$

We will now first consider the simple case of a nonterminal which is only visited once during an evaluation. When the node with structure s gets scheduled for evaluation the following steps are taken:

1. evaluate the values *ih* of the inherited attributes
2. if $s \times ih$ is in the domain of *M*, then share the attributed tree *M(s, ih)*.
3. if not create a node in *IH*, containing the values *s* and *ih* and visit that node.

In this way we achieve a *memo-function* implementation[Hug85]. Note that in this way all instantiations of a higher order attribute, when called with the same arguments, are shared and evaluated only once. An interesting aspect is now the following observation. When, as a result of reevaluation, a value in the *SH* changes, the nodes in *IH* referring to it will be marked for reevaluation. Because there are only as many nodes existent as there are different sets of inherited attributes a considerable increase in speed may be achieved. Not all instantiations, but only all different instantiations have to be reevaluated. Furthermore, those subtrees for which the reevaluation results in the same synthesized attributes as before, may be left out from the data structure maintained for rescheduling.

We now consider the case in which multiple visits may be necessary, and assume that the visits are ordered in some way. The solution is now to consider each visit as not only returning its associated synthesized attributes, but also the partially parameterised function corresponding to the as yet unevaluated part of the tree. This function is readily identified with the node in the *IH*, containing the inherited and synthesized attributes of the visit. So by extending the domain of the function *M* to $(SH \cup EH \times IH)$ and making the first component of the structures in the *IH* not only refer to elements in *SH* but also to elements in *IH*, the aforementioned memo-isation can be extended to the multiple visit case.

5 Conclusions

The correspondence of attribute grammars with functional programs has been indicated. We have shown how by using higher order attribute grammars, coupled with chaining nodes in the trees in order to find the appropriate places for performing the indicated transformations, the application area of attribute grammars can be considerably extended.

We expect that by further pursuing this similarity, and introducing some of the elegance which functional programs exhibit into the attribute arena, attribute grammars can be turned into a widely used programming formalism.

References

[Bir84] Richard S. Bird¡. The promotion and accumulation strategies in transformational programming. *TOPLAS*, 6(4):487–504, 1984.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* International Series in Computer Science. Prentice Hall, 1988.

[Chr89] Henning Christiansan. Structure sharing in incremental systems. *Structured Programming*, 10(4):169–186, 1989.

[CU77] J. Craig Cleaveland and Robert C. Uzgalis. *Grammars for Programming Languages.* Elsevier Computer Science Library, 1977.

[FZ88] Paul Franchi-Zannettacci. Attribute specifications for graphical interface generation. Rapport de Recherche 937, INRIA, Décembre 1988.

[GG84] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *SIGPLAN Notices*, pages 157–170. ACM, 1984.

[HK88] Scott E. Hudson and Roger King. Semantic feedback in the higgens uims. *IEEE Transactions on Software Engineering*, 14(8):1188–1206, August 1988.

[Hoo86] Roger Scott Hoover. Dynamically bypassing copy rule chains in attribute grammars. In *Proceedings 13th Annual ACM Symposium on Principles of Programming Languages*, pages 14–25, 1986.

[HT86] Susan Horwitz and Tim Teitelbaum. Generating editing environments based on relations and attributes. *TOPLAS*, 8(4):577–608, October 1986.

[Hug85] J. Hughes. Lazy memo-functions. In J-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 129–146. Springer, 1985.

[KBHG+87] B. Krieg-Brückner, B. Hoffmann, H. Ganzinger, M. Broy, R. Wilhelm, U. Möncke, B. Weisberger, A. McGettrick, I.G. Cambell, and G. Winterstein. Program development by specification and transformation. In *Proceedings of ESPRIT Conference 1986*. North-Holland, 1987.

[Knu68] D.E. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127–145, 1968.

[Knu71] D.E. Knuth. Semantics of context-free languages (correction). *Math. Syst. Theory*, 5(1):95–96, 1971.

[KS86] M.F. Kuiper and S.D. Swierstra. Using attribute grammars to derive efficient functional programs. Technical Report RUU-CS-86-16, Dept. of Computer Science, P.O.Box 80.089, 3508 TB Utrecht, the Netherlands, 1986.

[Mea86] James H. Morris and et al. Andrew: A distributed personal computing environment. *CACM*, 29(3):184–201, March 1986.

[PK82] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *TOPLAS*, 4(3):402–454, 1982.

[Rep82] Thomas Reps. Optimal-time incremental semantic analysis for syntax directed editors. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 169–176, January 1982.

[Rep84] Thomas Reps. *Generating Language Based Environments*. PhD thesis, M.I.T. Press, 1984.

[RMT86] Thomas Reps, Carla Marceau, and Tim Teitelbaum. Remote attribute updating for language based editors. In *13th Annual ACm Conference on the Principles of Programming Languages*, pages 1–13. ACM, 1986.

[RT89] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual.* Springer, third edition, 1989.

[RTD83] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language based editors. *TOPLAS*, 5(3):449–477, July 1983.

[SG86] Robert W. Schleifer and Jim Gettys. The x window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[Tur85] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 1–16. Springer, 1985.

[VBF90] Harald Vogt, Aswin van den Berg, and Arend Freije. Amazingly fast development of a program transformation system with attribute grammars and dynamic transformations. In *this conference??*, 1990.

[vE88] Peter H.J. van Eijk. *Software Tools for the Specification Language Lotos.* PhD thesis, Twente University, P.O.Box 217, 7500 AE Enschede, the Netherlands, 1988.

[VSK89] Harald H. Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher order attribute grammars. In *SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland, Oregon*, pages 131–145. ACM, 1989.

[Wea75] A. van Wijngaarden et. al. *Revised Report on the Algorithmic Language Algol 68.* Springer, 1975.

[Yeh83] Dashing Yeh. On incremental evaluation of ordered attributed grammars. *BIT*, 23:308–320, 1983.