TR 80-002

A Space Improvement in the Alternating Semantic Evaluatcr

Mehdi Jazayeri
Diane Pozefsky

A Space Improvement in the Alternating Semantic Evaluator

Mehdi Jazayeri
Department of Computer Science
University of North Carolina
Chapel Hill, N. C. 27514

Diane Pozefsky
IBM
Research Triangle Park, N. C. 27709

Abstract

It is possible to reduce the space requirements of pass-oriented attribute grammar evaluators by abandoning the traditional bias towards evaluating an attribute as early as possible. Two techniques for achieving this improvement are presented. Empirical data show the superiority of these techniques over traditional evaluation methods.

Keywords and phrases: alternating semantic evaluator, attribute grammars, lazy evaluation, storage optimization.

1. INTRODUCTION

The Alternating Semantic Evaluator (ASE) was introduced in [1] as a technique for evaluating attribute grammars [2]. An improvement in the evaluation strategy aimed at minimizing traversals of the parse tree was proposed and subsequently implemented by Raiha and Saarinen [3]. Here we introduce another optimization for ASE. The difference between our current proposal and most other attribute evaluation strategies in the literature is that we advocate delaying the evaluation of each attribute as long as possible. In contrast, the assumption behind all current evaluation strategies is that an attribute should be evaluated as soon as possible--that is, as soon as the attributes it depends on become available. We refer to our evaluators as _lazy evaluators_. Unlike Henderson and Morris's lazy evaluator for LISP [4], we don't save actual computations. The advantage we hope to gain is that by delaying the computation of an attribute we shorten the time that the storage for the attribute needs to be allocated and this in turn should reduce storage requirements for all attributes. The point we would like to emphasize is that the improvements of lazy evaluation are achieved at _no_ extra cost.

We will assume a basic knowledge of attribute grammars and ASE. In the next section we introduce the notation used in this paper. In section 3 we give two construction algorithms for lazy evaluators. It is interesting that only the construction algorithm is different from ASE; the evaluator algorithm is the same. In section 4 we give test results for comparing the performance of a lazy evaluator and a traditional evaluator in compilers for PL360, an Algol 60 subset, and two structured-FORTRAN preprocessors. In section 5 we offer some conclusions.

2. NOTATION

An attribute grammar is a context-free grammar with attributes associated with each nonterminal. After each production in the context free grammar, a number of semantic rules are given which define the values of the attributes of nonterminals of the production. The value of an attribute is defined as a function of the attributes that belong to the same production. We define the relation _dependson_ over the attributes. a _dependson_ b if in some semantic rule a is defined in terms of b. We will also use _dependson*_ which is the transitive closure of _dependson_.

An Alternating Semantic Evaluator evaluates all the attributes of a given parse tree by making m passes over that tree, evaluating the set A_i at each pass i. These passes may be either left-to-right (left) or right-to-left (right) over the parse tree. For each attribute x, pass(x) is the number of the pass on which x is evaluated. An ASE constructor produces the m sets, A_1 through A_m for a given grammar. Each set A_i consists of the attributes evaluable in pass i of any parse tree of the grammar.

For any two attributes x and y and pass i we define the relation occursbefore such that an attribute x occursbefore y if during the tree traversal of pass i, attribute x is visited before attribute y. The occursbefore relation depends on the direction of the pass. Since the traversal order of each pass is fixed, the occursbefore relation is known at construction time for each grammar.

3. CONSTRUCTION ALGORITHMS

The original ASE construction algorithm, which we will henceforth call forward ASE, starts with the initial assumption that all attributes are evaluable on the first pass. These attributes are thus placed in set A_1. Examination of the semantic rules of the grammar forces the elimination of some attributes from this set. These are then assumed to belong to A_2. This process is repeated until all attributes have been assigned to a set A_i. The following is the part of the algorithm which forms the set A_i for pass i.

```
for each attribute x in $A_i$
     verify that for every attribute y such that x dependson
     y, pass(y)<i or (pass(y)=i and y occursbefore x)
```

```
    otherwise delete x and all attributes z such that z
    dependson* x from $A_i$ and insert them in $A_{i+1}$.
```

We now give two methods for producing lazy evaluators.

3.1 BACKWARD SELECTION

The first method which we call backward selection is a simple transformation of the original algorithm. Instead of assuming initially that all attributes are evaluable on the first pass and deleting those that are found not to be, we assume all attributes are evaluable on the last pass and delete those that are needed before the last pass. Because the number of required passes is not known in advance, some postprocessing is needed to renumber the passes in reverse order of their production. In this algorithm, pass "1" is the first pass produced by the constructor but the last pass for the evaluator. Renumbering the passes is a simple task and requires O(A) steps where A is the number of attributes.

The algorithm section above may be restated thus:

```
Backward ASE
for each attribute x in A
    verify that for every attribute y such that y dependson
    x, pass(y)<i or (pass(y)=i and x occursbefore y)
    otherwise delete from $A_i$, the attribute x and all
    attributes z such that x dependson* z and insert them
    in $A_{i+1}$.
```

The postprocessing is simple:

renumber each set A_i, to A_{m-i+1}

3.2 PASS COMPRESSION

The second method of producing lazy evaluators is called pass compression and is considerably more complicated. The basic idea is that first the original construction algorithm is used and m sets A_i are produced. Then, in a postprocessing step, an attempt is made to move an attribute definition forward (toward A_m) as close as possible to its first use. An attribute can be moved to the pass on which it is first used or to the pass prior to it, depending on the occursbefore relation.

In order to move each attribute only once, we want the final position of the dependent attribute y to be defined before we attempt to move the attribute x that dependson y. In order to do this, we want to process the later passes first, that is, process the m passes in reverse order.

Within a pass, however, if a dependson b we must process a before b lest we miss a possibly useful movement. That is, we can move a to a later pass and then move b up to it; if we attempt to move b first, it would be blocked by a's dependence on it. One would therefore like to perform a topological sort on the attributes asigned to a pass. This, however, is not possible. Look, for example, at the dependency graph segment of Figure 1.

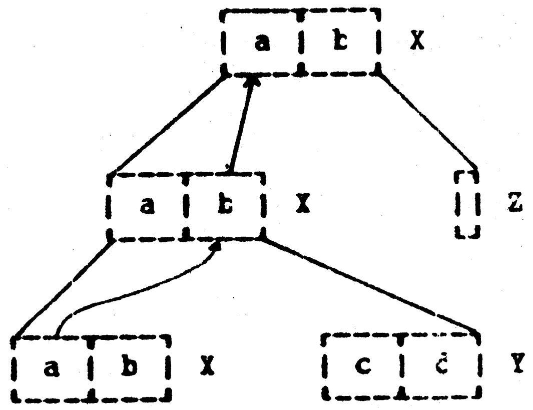

Figure 1. Dependency graph of a ring

The arrows indicate the flow of information; if a <u>dependson</u> b, then the arrow will point from b to a. a(X) and b(X) clearly must be evaluated on the same pass. Further, we have both a <u>dependson</u> b and b <u>dependson</u> a. When we have a group of attributes that show this type of dependence, the movement decision will have to be made for the entire group together.

We refer to such a group as a <u>ring</u>. Formally, a set of attributes $x_1, x_2, \ldots x_n$ form a ring <u>iff</u> for all i, $1<i<n-1$,

(x_i <u>dependson*</u> x_{i+1} and x_{i+1} <u>dependson*</u> x_i). A bit matrix representation of <u>dependson*</u> will show these attributes as having identical rows and columns. This matrix is also helpful in ordering the attributes of a pass for processing: by sorting the attributes by the total number of attributes that they depend on (number of ones in a row of the matrix), we can determine which attributes must be processed first (see below).

Once we have determined the members of the ring, we move all its elements to the earliest pass that every one can be moved to. Before we give the postprocessing algorithm, we offer some further clarifying remarks.

Since we are moving attributes as late as possible, there is no processing required of the final pass; therefore the outer loop begins with the next to last pass. Attributes are ordered and processed with those with the most dependencies first. The reason for this order is that if <u>a</u> dependson <u>b</u>, they are not part of a ring, and they are evaluated on the same pass, we want to attempt to move <u>a</u> and then <u>b</u>, since <u>b</u> could not be moved to a pass later than <u>a</u>. Clearly <u>a</u> has at least one more dependency than <u>b</u> so the decreasing number of dependencies is the correct order. This is normally a method that can be used to get a topological sort on the relation transpose(dependson). As we pointed out earlier, a topological sort would fail, but the algorithm does give us a valid ordering. After this much of the sort has been done, the algorithm can identify rings by a pair-wise comparison between all possible attributes with the same number of dependencies.

```
Algorithm Pass-Compression:
 /* process each pass,P */
for P := N-1 to 1 by -1
do
     sort the  attributes  to  be evaluated  on  pass  P  by
          decreasing number of dependencies;

     identify any rings within the groups of attributes with
          the same number of dependencies;

     for each attribute x of the pass in order

     do
          if x is not a member of a ring

          then   /* process a single attribute */
               find the set Y = {y|y dependson x};
               m = $\min_{y \in Y}$ (pass(y));
               if x occursbefore  y for  all y $\in$ Y such  that
                    pass(y) = m

               then move x to pass m

               else move x to pass m-1

               fi

          else
               if x is the first member  of the ring encoun-
                    tered

               then  /* process entire ring */
                    find the set  Y = {y|y dependson  x' and
                         x' is  a member  of the  ring being
                         processed}

                    if x' occursbefore  y for all x'  in the
                         ring and all y  Y such that pass(y)
                         = m
                    then move all ring members to pass m

                    else move all ring members to pass m-1

                    fi
               fi
          fi
     od
od
```

The cost of the algorithm is dominated by the inspection of the <u>dependson*</u> relation and is $O(A^2)$ where A is the total number of attributes in the grammar. Note the complexity of the ASE algorithm is $O(A^3)$ and the <u>dependson*</u> relation needs to be available for the construction algorithm anyway.

4. EMPIRICAL DATA

The motivation for using lazy evaluators is to save space by delaying the allocation of storage for attributes. We have run some sample grammars to measure the possible savings and also the difference between the performance of the pass-compression and the backward selection algorithms.

In order to compare the performances of the different algorithms we need to define the criteria. We make the following definitions.

The <u>pass-life of an attribute</u> is m-n+1 where m is the number of the pass on which the attribute is used for the last time, and n is the number of the pass on which the attribute is evaluated. The pass-life of an attribute indicates the minimum number of passes during which storage must be allocated for the attribute.

The <u>average pass-life of a pass selection for an attribute grammar</u> is defined as the average of the pass-lives for all its attributes. Two possible pass selections for the same attribute grammar may be compared by their pass-lives because a lower number indicates that less storage is required.

Another useful measure for a particular pass selection is the _maximum number of active attributes_, where an attribute is _active_ from the pass it is defined on until the pass in which it is last used. The reason for using this measure is that it is not necessarily the time-space product that we want to minimize, as the average pass life would, but the maximum space required. It should be noted that more accurate figures would take into account the relative occurrences of nonterminals and their attributes in parse trees. To our knowledge, such figures are not available.

The pass-compression algorithm is quite expensive to run and we have only tested it on small grammars. On these cases, only once was the performance of the pass compression different from backward selection in the number of passes: it required one less pass (i.e., all of the attributes were moved from one pass). In the other cases, the number of passes were the same but the characteristics of the passes were different. When there is a difference, backward selection gives lower average pass life (better time-space product) and pass compression gives smaller maximum number of active attributes (lower max storage).

We have done a much more thorough comparison of the forward and backward selection algorithms. In addition to small example grammars that were used for examination of pass-compression, five sizable grammars were also compared for their storage requirements under the forward and backward ASE schemes. These grammars were: two different versions of S-FORTRAN (a structured FORTRAN preprocessor), an ALGOL-60 subset, and two versions of PL360. Figures 2 and 3 show the results of the comparison for these large grammars.

These figures need little interpretation. The first table shows that the space-time product for ASE is higher than for lazy ASE (LASE). The second table shows that the maximum number of attributes that need to be stored during evaluation are less with LASE than with ASE-- sometimes considerably less. These figures together confirm our expectation that memory requirements for lazy evaluation are less than for original ASE.

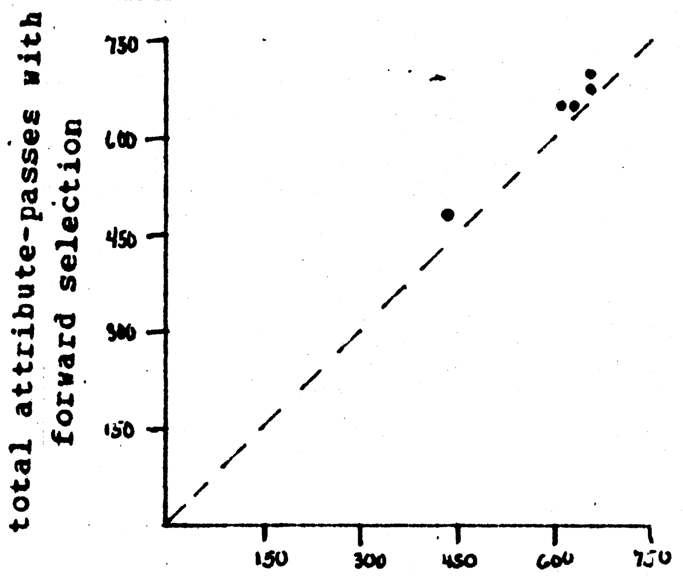

Figure 2: Attribute passes for complete grammars

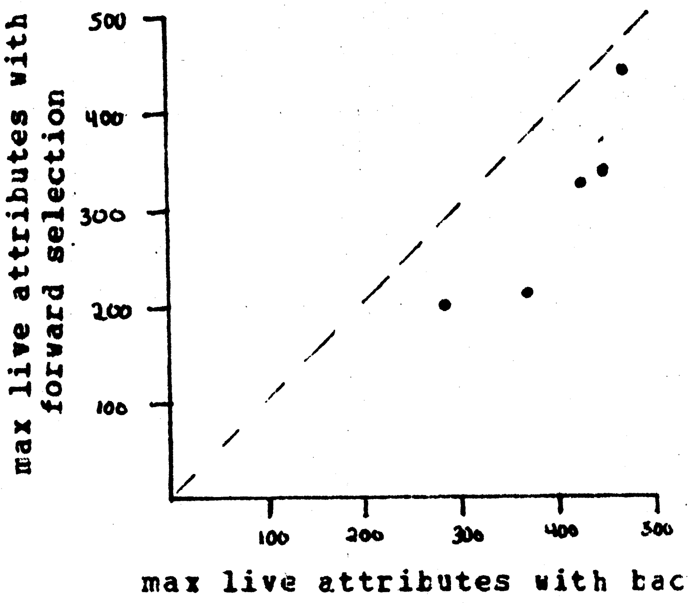

Figure 3: Max live attributes for complete grammars

5. CONCLUSIONS AND DISCUSSION

The universal assumption made with respect to attribute grammar evaluators is that one must evaluate an attribute as soon as possible. This assumption is not valid if one is interested in economy of space during evaluation of the grammar. We have presented two methods for converting an existing evaluator, ASE, into one that works according to the assumption that an attribute should be evaluated as late as possible. We presented experimental data that indicate that these so-called lazy evaluators are indeed capable of reducing storage requirements.

Many interesting questions are suggested by this work. What is the exact relationship between backward selection and pass compression? For example, when will they produce different numbers of passes? Is the extra cost of pass compression ever justified? We suspect not. We believe that the difference stems more from the direction (left or right) of the first pass selected. Can lazy evaluation be applied to other evaluation strategies such as that of [5]? Are the comparison criteria we have used adequate? Some of the deficiencies with these measures is discussed in [6] but can we can do better with more sophisticated measures?

Whatever the answers to these questions might be, the key point of this paper is that backward selection produces better results than forward selection and this improvement is gained at <u>no</u> extra cost. The forward selection algorithm which is implemented in several systems should, therefore, be used only if an external reason exists.

Acknowledgements

The large grammars used in this study were obtained from the Computer Science Department of the University of Helsinki. This work was supported in part by the National Science Foundation Grant No. MCS77-03729.

REFERENCES

[1] Jazayeri, M. and K.G. Walter, "Alternating Semantic Evaluator", Proceedings of the ACM Annual Conference, Minneapolis, October 1975.

[2] Knuth, D.E., "Semantics of Context Free Languages", Mathematical Systems Theory 2, 2, 1968.

[3] Raiha, K-J. and M. Saarinen, "An Optimization of the Alternating Semantic Evaluator", Information Processing Letters 6, 3, June 1977.

[4] Henderson, P. and Morris, J.H., "A Lazy Evaluator," Communications of the ACM, Volume 4, Number 1, January 1961, pp. 51-55.

[5] Kennedy, K. and S. K. Warren, "Automatic Generation of Efficient Evaluators for Attribute Grammars", Conference Record of the Third ACM Symposium on Principles of Programming Languages, Atlanta, January 1976.

[6] Pozefsky, D., "Building Efficient Pass-Oriented Attribute Grammar Evaluators", Ph.D. Dissertation, Computer Science Department, University of North Carolina, April 1979 (UNC TR-79-006).