Efficient Evaluation of Well-Formed Attribute Grammars And Beyond

J. Grosch

DR. JOSEF GROSCH

COCOLAB - DATENVERARBEITUNG

GERMANY

Cocktail

Toolbox for Compiler Construction

Efficient Evaluation of Well-Formed Attribute Grammars And Beyond

Josef Grosch

Oct. 15, 1992

Document No. 31

Dr. Josef Grosch
CoCoLab - Datenverarbeitung
Breslauer Str. 64c
76139 Karlsruhe
Germany

Phone: +49-721-91537544
Fax: +49-721-91537543
Email: grosch@cocolab.com

Efficient Evaluation of Well-Formed Attribute Grammars And Beyond

Abstract

This paper is concerned with the evaluation of well-formed attribute grammars as support for semantic analysis. This is the largest class of attribute grammars, which does not impose any restrictions. A data structure and an algorithm are described which are efficient, both in terms of space and time. Attribute grammars are extended from the decoration of trees to the decoration of graphs and to the access of non-local attributes. Experimental results are presented that compare the space and time behaviour of evaluators for well-formed attribute grammars and ordered attribute grammars.

Keywords well-formed attribute grammars, attribute evaluation

1. Introduction

One possibility to support the definition and implementation of semantic analysis in a systematic and formal manner is the use of an attribute grammar [Knu68]. From this kind of specification, attribute evaluators that implement semantic analysis can be generated automatically by appropriate tools. After 25 years of research in the area of attribute grammars, tools that process this kind of specification are still not as popular as e. g. scanner and parser generators. Some reasons for this are that in the past attribute evaluators consumed too much memory and too much run time or they handled restricted classes of attribute grammars, only. Now even well-formed attribute grammars, the largest class of attribute grammars, can be evaluated efficiently, as will be shown in this paper.

Pure attribute grammars advocate a functional style. Computation rules describe the computation of attribute values by expressions that may depend on other attribute values. All those dependencies must be given explicitly. There are neither global attributes nor side-effects. These functional dependencies are the basis for checks that can be performed automatically. Attribute grammars can be checked for the correct use of synthesized and inherited attributes, for completeness, and for the absence of cyclic dependencies. An attribute grammar is complete if there are computation rules for all attributes. An attribute grammar has cyclic dependencies if there are attributes that depend on themselves, either directly or indirectly. Furthermore, the dependencies among the attributes allow the determination of an evaluation order for the attributes. While the checks that can be performed on an attribute grammar are certainly valuable, the automatic determination of an evaluation order is attractive only at the first glance. It works well for small applications with today's technology.

However, our experience with large applications using state of the art attribute grammar tools shows that there is no real relief of the burden to worry about the evaluation order. Our recent projects involved attribute grammars for Modula-2 [Mar90], Modula-3 [Kie91], and the robot programming language IRL [IRL92]. Today's attribute grammar tools usually handle restricted classes of attribute grammars such as for example ordered attribute grammars [Kas80] where the evaluation order can be determined statically during generation time. It is argued that those restrictions are necessary in order to achieve efficient attribute evaluation. In larger applications often cyclic dependencies among the attributes are introduced by the closure algorithms which are due to those restricted classes of attribute grammars. In order to remove those cyclic dependencies it is necessary to worry quite a lot about evaluation order and to introduce additional attributes along with additional (maybe dummy) computations. These steps solve the problems with the cycles but lead to rather complicated attribute grammars. In our experience, those grammars degrade from formal problem specifications to problem implementations which are hard to understand and to maintain [Mar90].

From the drawbacks mentioned above we conclude that substantial improvements are necessary in the area of attribute grammars. Therefore we propose to use the class of well-formed attribute grammars, which is the largest class of useful attribute grammars. An attribute grammar is well-formed or non circular if for every tree the dependency relation among all attributes is cycle free. This approach retains the advantages of attribute grammars mentioned before such as formal consistency checks and the automatic determination of the order of the computations. Smaller classes such as ordered attribute grammars are considered to be too restrictive because one has still to bother with the evaluation order. Naturally, evaluators for well-formed attribute grammars are not as efficient as those for e. g. ordered attribute grammars. For ordered attribute grammars the evaluation order (visit sequence) can be determined statically during the generation of the evaluator. For well-formed attribute grammars the evaluation order must be determined dynamically during the run time of the evaluator. Of course, there is a trade-off between expressive power and efficiency. This paper shows that well-formed attribute grammars can be evaluated efficiently both in terms of space and time. Compared to ordered attribute grammars the increased consumption of memory and run time is relatively small and it is tolerable with today’s hardware.

Furthermore, the attribution of trees will be extended to the attribution of graphs. While conventional attribute grammars base computations only on attributes local to a grammar rule, we will allow the access of non-local attributes. The processing of graphs and access to non-local attributes is of interest for example for the handling of symbol tables in an attribute grammar style. With the latter features the foundations of conventional attribute grammars are left and this explains the word *beyond* in the title. It turns out that these features fit together nicely with the evaluation of well-formed attribute grammars.

This paper is organized as follows: Section 2 describes the data structure of the trees to be decorated with attributes. Section 3 presents the principle of the algorithm for the evaluation of well-formed attribute grammars. Section 4 applies several transformation steps to this algorithm using partial evaluation and turns it into a more efficient version. Section 5 describes the extensions concerning graphs and access to non-local attributes. Section 6 reports experimental results. Section 7 describes related research. Section 8 contains concluding remarks.

2. Data Structure

They key idea of our approach is to use a recursive attribute evaluator that computes attributes by need [DJL88]. The run time of such an evaluator is theoretically time-optimal. It is sublinear in the size of the tree because it computes only those attributes necessary for the final result. However, the stack space is not limited by the height of the tree. We implement a procedure C (t, i) which computes the i-th attribute of a rule that is represented by a tree node at the address t and which stores the result in the tree. Before the computation of an attribute, the attributes it depends on are computed by appropriate recursive calls of this procedure. Afterwards its value can be computed and stored, because all information necessary is locally available within the context of the current rule. One attribute may be used several times but of course it suffices to compute it only once.

There are basically two possibilities for the underlying data structure: One can either use the (abstract) syntax tree or the graph of dependencies among the attributes. We have chosen the first alternative because it promises to be space efficient. The properties of the data structure are enumerated first and explained in more detail afterwards. Fig. 1 illustrates the internal representation of a grammar rule such as $X_0 = X_1\ X_2\ X_3$ in the tree. The node n_0 stands for the left-hand side and the nodes n_1, ..., n_3 stand for the right-hand side. In this example it is assumed that every node has three children and four attributes. The attributes of all nodes n_0, ..., n_3 would be accessible in the context of this rule.

1. Every tree node has pointers referring to all of its children or subtrees.
2. Every tree node has a pointer referring to its parent node.
3. All attributes are stored in the tree.
4. For every attribute there is a bit that indicates whether the attribute is already computed.
5. All attribute occurrences within a rule are numbered (see below).
6. Every tree node stores an offset that allows to convert attribute numbers between different rule contexts.

The pure syntax tree (1) is not sufficient. It must be augmented by pointers to parent nodes (2), indicator bits (4), and certain offsets (6) in order to avoid unnecessary recomputations of attributes and to enable recursive calls that climb up in the tree. The attribute dependencies are contained implicitly in this structure together with the knowledge about the dependencies available during generation time.

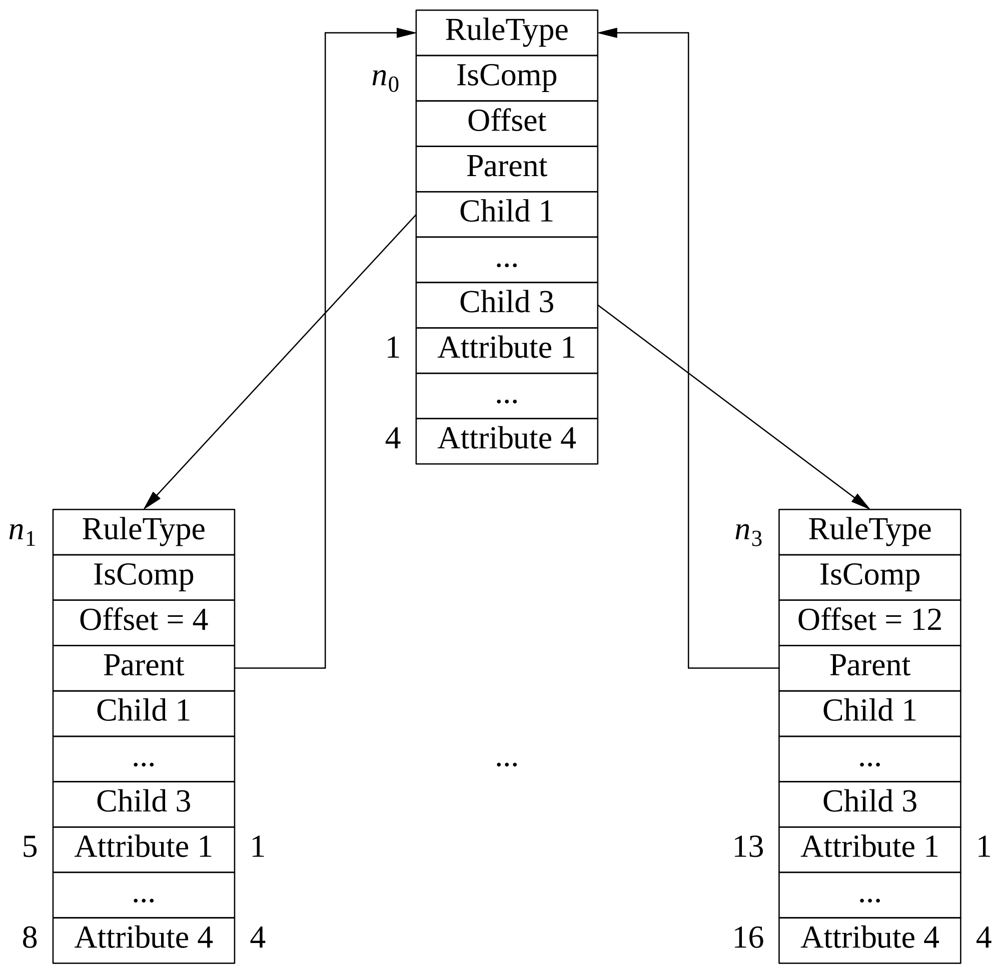

Fig. 1: Data Structure

The pointers to parent nodes (2) are necessary for the computation of inherited attributes which are performed in the context of the parent rules.

The chosen representation has the disadvantage that optimization of attribute storage is impossible and therefore all attributes have to be stored in the tree (3). However, this imposes no restriction with today’s memory capacities. The use of a complete dependency graph would allow an optimization that replaces several attribute instances whose values are copies of each other by a single instance.

As already mentioned, a bit is maintained for every attribute (4) in order to evaluate the attribute value only once. All the bits for one node are combined in a bit vector called *IsComp*.

All attribute occurrences of the left-hand side and the right-hand side of a rule are numbered (5). For example, in Fig. 1 the attributes in the context of the tree node n_0 are numbered from 1 to 16 as given by the numbers to the left of the attribute fields.

When the context is changed from a parent node to child node or vice versa then right-hand side attributes become left-hand side attributes or vice versa. In the new context the attribute numbers are different. The offset (6) represents the difference between those two attribute numbers. For example, if the context changes from the node n_0 to one of its children n_1 or n_3 then the attributes of n_1 or n_3 would receive the numbers given to the right of the attribute fields.

The data structure for the tree is built in two phases. First, space is allocated for the tree nodes and pointers to the child nodes are established. This usually happens during parsing. Second, before attribute evaluation actually starts, a traversal of the tree is performed that initializes the parent pointers, the indicator bits, and the offsets.

3. Basic Algorithm

The procedure C (t, i) as the core of an attribute evaluator is generated from an attribute grammar. It constitutes a directly coded evaluator rather than a table-driven one. It will compute the attribute with the number i at a tree node with the address t. The general structure is shown in Fig. 2 using a C-like notation. It consists of two nested switch statements that discriminate the present rule type and attribute number. For all those cases there will be a code segment which is structured as the one given in Fig. 2. First, the indicator bit is checked whether this attribute has already been computed. Then the availability of the attributes used as arguments for the current computation is assured by appropriate recursive calls of the procedure C. Then the attribute is computed and stored in the tree. Finally, the bit is set to indicate that the attribute has been computed.

The code scheme in Fig. 2 uses two macros: IN (b, v) tests whether bit number b is set in the bit-vector v. INCL (v, b) sets bit number b in the bit-vector v. The selector names *Child* and *Attribute* would be replaced by specific names to access child nodes and attributes. Most of the attribute numbers are constants whose values can be determined at generation time (i, j, k, l).

In simple cases the result of an attribute evaluation are the values of the synthesized attributes at the root node. In this case the evaluation would be started by appropriate calls of the procedure C for the synthesized attributes of the root. In general, one is interested in the values of so-called output attributes which may be spread all over the tree. Output attributes are those attributes that are supposed to hold their value upon termination of attribute evaluation. For example, the tree and its output attributes might constitute the input to a succeeding tree transformation using a tool based on pattern-matching such as *puma* [Gro92]. Moreover, an attribute evaluator has to check conditions of the attribute values. Conditions can be regarded as boolean output attributes whose values do not have to be stored. The above considerations explain the necessity of an evaluator routine E that

```
void C (tTree t, int ii)
{
  switch (t->RuleType) {
  ...
  case n:   /* one case for every rule type n                                 */
    switch (ii) {
    ...
    case i: /* one case for every attribute number i of rule type n
               that is to be computed in the context of this rule type       */

            /* check whether i is already computed:                           */
      if (IN (i, t->IsComp)) return; /* if i is attribute of left-hand  side */
      if (IN (l, t->Child->IsComp)) return; /* l = i - t->Child->Offset      */
                                     /* if i is attribute of right-hand side */

            /* one call for every attribute j that attribute i depends on:    */
      C (t, j);          /* if j is synthesized attribute of left-hand  side */
      C (t->Parent, t->Offset + j);
                         /* if j is inherited   attribute of left-hand  side */
      C (t->Child, k);   /* k = j - t->Child->Offset                          */
                         /* if j is synthesized attribute of right-hand side */
      C (t, j);          /* if j is inherited   attribute of right-hand side */

            /* compute attribute i:                                           */
      t->Attribute = ...             /* if i is attribute of left-hand  side */
      t->Child->Attribute = ...      /* if i is attribute of right-hand side */

            /* mark attribute i as computed:                                  */
      INCL (t->IsComp, i);           /* if i is attribute of left-hand  side */
      INCL (t->Child->IsComp, l);    /* if i is attribute of right-hand side */
      break;
    ...
    }
  ...
  }
}
```

Fig. 2: Basic Algorithm

traverses those subtrees that may contain output attributes or conditions and that triggers their evaluation by appropriate calls of the procedure C. The procedure E can be optimized using tail recursion.

The presented version of the procedure C works, however, it has a few drawbacks. In real applications it could become tremendously large and it may exceed compiler restrictions. Furthermore, it can be made more efficient as shown in the next section.

4. Optimized Algorithm

Several transformation steps can be applied to the basic algorithm of the previous section resulting in a more handy and efficient version:

First, the procedure C can be split in two procedures I and S that handle inherited and synthesized attributes separately. This reduces the procedure size to around the half.

Second, the procedures I and S can be split into two sets of procedures with elements I_i and S_i. This is done by applying partial evaluation to I and S with respect to the second arguments for all existing attribute numbers. This step drastically reduces the procedure sizes. Only one argument

```
typedef void (* tP) (tTree);
tP I [] = { 0, I_1, I_2, ... };

void I_i (tTree t)   /* one procedure for every inherited attribute number i */
{
  switch (t->RuleType) {
  ...
  case n:                                 /* one case for every rule type n */

            /* one call for every attribute j that attribute i depends on:   */
    if (! IN (j, t->IsComp)) S_j (t);
                         /* if j is synthesized attribute of left-hand  side */
    if (! IN (j, t->IsComp)) I [t->Offset + j](t->Parent);
                         /* if j is inherited   attribute of left-hand  side */
    if (! IN (k, t->Child->IsComp)) S_k (t->Child);/* k=j - t->Child->Offset */
                         /* if j is synthesized attribute of right-hand side */
    if (! IN (k, t->Child->IsComp)) I_j (t);
                         /* if j is inherited   attribute of right-hand side */

    t->Child->Attribute = ...             /* compute attribute i            */
    INCL (t->Child->IsComp, l);           /* mark attribute i as computed   */
    break;                                /* l = i - t->Child->Offset       */
  ...
  }
}

void S_i (tTree t) /* one procedure for every synthesized attribute number i */
{
  switch (t->RuleType) {
  ...
  case n:                                 /* one case for every rule type n */

            /* one call for every attribute j that attribute i depends on:   */
    if (! IN (j, t->IsComp)) S_j (t);
                         /* if j is synthesized attribute of left-hand  side */
    if (! IN (j, t->IsComp)) I [t->Offset + j](t->Parent);
                         /* if j is inherited   attribute of left-hand  side */
    if (! IN (k, t->Child->IsComp)) S_k (t->Child);/* k=j - t->Child->Offset */
                         /* if j is synthesized attribute of right-hand side */
    if (! IN (k, t->Child->IsComp)) I_j (t);
                         /* if j is inherited   attribute of right-hand side */

    t->Attribute = ...                    /* compute attribute i            */
    break;
  ...
  }
  INCL (t->IsComp, i);                    /* mark attribute i as computed   */
}
```

Fig. 3: Optimized Algorithm

needs to be passed to the procedures instead of two. One of the two switch statements disappears. In some cases the remaining switch statement contains only one case and then it can be removed, too.

In all cases but one it is statically known which procedure of the two sets has to be called. In the case of an inherited attribute of the left-hand side the procedure to be called depends on the offset of the current node which is known only during run time. This can be solved by an array of procedures I which is initialized such that I [i] refers to I_i. Then those calls access the array with a dynamically computed index in order to determine the required procedure. The calls have the following form:

```
    I [t->Offset + j] (t->Parent);
```

Third, the check whether an attribute is already computed can be moved from inside the procedures to all the places immediately before the call of the procedures. This increases code size but decreases run time by avoiding unnecessary procedure calls if there is more than one use of an attribute.

Fourth, the handling of conditions can be optimized in several respects: No indicator bits are necessary for conditions because the results are neither stored nor can they be used. The calls of the procedures S_i can be saved if the code segments for conditions are moved from the procedures S_i into the evaluator routine E. If the conditions are checked after the evaluation of the output attributes it suffices to trigger the evaluation of the non-output attributes used as arguments in the conditions. If there are several conditions for one rule type and some use one non-output attribute as argument then it suffices to trigger its evaluation only once.

The resulting structure of the procedures I_i and S_i is presented in Fig. 3.

5. Extensions to Attribute Grammars

Conventionally, attribute evaluation is performed on trees. It can be extended to handle graphs as well, if several problems are suitably solved.

First, the driver routine for attribute evaluation E has to handle at least strongly connected, directed graphs. In order to avoid infinite loops arising from cycles in the graph a depth first traversal of the graph can be performed using a simple algorithm with a mark bit.

Second, in graphs a node can have several parent nodes. If there are synthesized attributes then it suffices to compute them upon the first visit coming from one of the parent nodes. When further visits coming from other parents require this value it needs not to be computed again and this is assured by the mechanism with the indicator bit. Here attribute evaluation on graphs and our evaluator for well-formed attribute grammars fit together, nicely.

If there are inherited attributes then it must be decided which of the parent nodes should compute the value. We defined to select one as the "real" parent and thus being responsible for the evaluation of inherited attributes. The real parent node is selected during the initialization phase when the parent pointers are established.

Third, our implementation supports higher-order attribute grammars (HAGs) [VSK89, Vog93] which allow tree-valued attributes (pointers to subtrees) and that subtrees may be created dynamically during attribute evaluation. This can be realized relatively simple by treating pointers to children like attributes and by including them in the dependency analysis. In combination with the evaluation of well-formed attribute grammars it is necessary to initialize dynamically created subtrees with parent pointers, offests, and indicator bits.

Fourth, the HAG philosophy leads to the access of non-local attributes. For example, one possibility for the implementation of a symbol table is to use the abstract syntax tree itself as data structure. A symbol table entry would be accessed via a pointer to the node representing a declaration. At the applied occurrence of an identifier a lookup in the symbol table is performed that yields an attribute which is a pointer to a declaration. Of course, one is interested in accessing properties of this symbol table entry which is equivalent to the access of attributes of a node that is outside the context of the current rule. We defined the following syntax for this non-local attribute access:

```
    REMOTE address => rule_type : attribute
```

It specifies the address of a tree node which is supposed to have a certain rule type and the name of the attribute. Again this feature can be nicely implemented in our evaluator for well-formed attribute grammars. The rule type and the attribute name allow to determine the evaluation procedure and the address supplies the argument to the call. This call assures that the attribute is evaluated. Then its value can be accessed by dereferencing the pointer (address) appropriately. The dynamic behaviour of the evaluator allows the computation of non-local attributes without any particular extensions because it can be requested to compute any attribute.

6. Experimental Results

The algorithm described in the previous sections has been implemented by extending the generator for attribute evaluators *ag* [Gro] to produce evaluators for well-formed attribute grammars. The tool *ag* is part of a Toolbox for Compiler Construction called *Cocktail* [GrE90]. The tool statically checks that an attribute grammar is well-formed [RaS82] using an algorithm that incorporates the optimizations described in [DJL84, JoP88]. Although this test was proved to be intrinsically exponential in time and space [JOR75a, JOR75b] it showed to execute in reasonable time in practical cases. This has been observed by other authors, too [Aug90]. Currently this test assumes that attribution is performed on a tree. It disregards graphs and cyclic dependencies that can be introduced through the access of non-local attributes. In order to guarantee termination a dynamic check for cyclic dependencies can be generated, optionally.

In this section evaluators for well-formed attribute grammars (WAG) and for ordered attribute grammars (OAG) are compared with respect to their space and time behaviour. The evaluators for ordered attribute grammars were generated in two versions: with and without optimization of attribute storage. The evaluators for ordered attribute grammars generated by *ag* are claimed to be very efficient. Two attribute grammars have been used in the experiments: Grammar 1 has 381 lines and 195 attribute computation rules (including conditions), grammar 2 has 362 lines and 106 rules. Only 113 rules and 65 rules, respectively, are specified explicitly, the others are added automatically by the system as copy rules. Grammar 2 was derived from grammar 1 by two modifications: Instead of a separate data structure for the symbol table the abstract syntax tree was used for this purpose. The computation of label values for jump statements was moved from the attribute grammar to the intermediate code generator. The evaluators implement the semantic analysis for a demo language called *Minilax* which is a subset of Pascal. Besides semantic analysis, the complete experiment involved a front-end compiler for this language including lexical analysis, syntax analysis, abstract syntax tree construction, and the generation of an intermediate code.

The results of six experiments are summarized in Table 1. Line a gives the ratio of the results obtained by experiment 3 relatively to experiment 1 and line b gives the ratio of the results obtained by experiment 6 relatively to experiment 4. The results were obtained on a SPARC station 10 using as input a 8000 line Minilax program. The implementation language is C and the sources have been compiled with the command "cc -O". We measured the CPU times for the three phases of the front-end (Parse: scanning + parsing + tree construction, Eval: attribute evaluation, I-Code: intermediate code generation), the consumption of memory for heap and stack, as well as the sizes of the attribute evaluators.

In the case of the WAG evaluator for grammar 1 more resources are used in comparison to the OAG evaluator: Run time is increased by a factor of 4.8, heap space by 1.9, stack space by 1.2, and the size of the binary evaluator by 3.4. The total run time of the front-end increases by a factor of 1.9. In the case of the WAG evaluator for grammar 2 which has the same functionality as grammar

Table 1: Experimental Results

#	evalu-ator	gram-mar	opt.	run time [sec] Parse	Eval	I-Code	total	speed [lines/sec]	memory [KB] heap	stack	evaluator size [lines]	[KB]
1	OAG	1	yes	0.58	0.18	0.09	0.85	9421	789	16.1	911	6.9
2	OAG	1	no	0.62	0.21	0.11	0.94	8519	1240	11.5	730	6.3
3	WAG	1	-	0.67	0.87	0.11	1.65	4853	1506	18.8	1808	23.8
a	ratio 3/1			1.2	4.8	1.2	1.9	0.5	1.9	1.2	2.0	3.4
4	OAG	2	yes	0.58	0.16	0.12	0.86	9311	789	11.5	813	6.4
5	OAG	2	no	0.57	0.16	0.13	0.86	9311	809	11.5	609	5.0
6	WAG	2	-	0.63	0.36	0.12	1.11	7214	1076	17.5	1236	16.2
b	ratio 6/4			1.1	2.3	1.0	1.3	0.8	1.4	1.5	1.5	2.5

1 the increases are in general not so high: Run time is increased by a factor of 2.3, heap space by 1.4, stack space by 1.5, and the size of the binary evaluator by 2.5. The total run time of the front-end increases by a factor of 1.3. Although the phase Parse was identical in all experiments its run time varies because of the different amounts of heap memory that are allocated for the tree. For a complete compiler, the increase in the total run time will be smaller depending on the time need by the code generator. A WAG evaluator consumes more resources than an OAG evaluator. However, nowadays we do have those resources and the execution speed is still very high (7214 lines/sec). We are optimistic to reach the performance of OAG evaluators or to do even better, because well-formed attribute grammars are much easier to write and involve less attributes.

7. Related Research

During the past 20 years several generators for well-formed attribute grammars have been constructed: AG, CGSG, CIS, COPS, DELTA, Elegant, FNC/ERN, FOLDS, LINGUA, NEATS, RÜGEN, Synthesizer Generator, TALLINN. The survey book on attribute grammars [DJL88] contains many references to those systems. To our knowledge, only the systems DELTA, LINGUA and Elegant statically check whether an attribute grammars is well-formed. Some of the other systems check this property dynamically during attribute evaluation. The underlying data structure is either the syntax tree or the associated graph of dependencies among the attributes. Almost all attribute evaluators are rather slow and consume huge amounts of memory especially if the complete dependency graph is constructed.

The Elegant system [Aug90] is remarkable because it is reasonable efficient. Front-ends generated by Elegant run at a speed of 600 to 820 lines/sec on a SUN 4/390 and need 500 KB of memory for 1000 lines of input text. The rather large consumption of memory is due to the complete dependency graph. According to the author this limits the use of compilers generated by Elegant to modules of a few thousand lines. The system can handle so-called pseudo circular attribute grammars and supports the manipulation of unevaluated attributes by a lazy evaluation technique. If we do our experiment on a SUN 4 the *Minilax* front-end would run at a speed of 5000 lines/sec and consume 135 KB for 1000 lines. Even if these numbers might change for real applications by a factor of up to two then still a big gain of efficiency has been achieved by our implementation scheme.

8. Conclusion

The presented implementation scheme allows for the generation of evaluators for well-formed or non circular attribute grammars that are efficient, both in terms of space and time. It is no longer necessary to use restricted classes of attribute grammars. Moreover, attribution of trees has been extended in several ways including attribution of graphs, higher order attribute grammars, and access to non-local attributes.

The efficiency of the attribute evaluators makes the attribute grammar technology attractive for real world applications that demand for production quality. The unrestricted class of well-formed attribute grammars together with the described extensions can increase the acceptance of attribute grammars as specification technique for semantic analysis.

Our generator for evaluators of well-formed attribute grammars is an available product. It is implemented in C and generates evaluators in C, C++, or Modula-2. It is portable and known to run on commonly used operating systems such as UNIX and MS-DOS.

Future work will include alternative implementation schemes such as a table-driven evaluator, the extension of the test for well-formedness to graphs and to non-local attribute access, performance analysis for a large application, and the development of a style guide that exploits the possibilities of well-formed attribute grammars and the presented extensions.

References

[Aug90] L. Augusteijn, The Elegant Compiler Generator System, *LNCS 461*, (Sep. 1990), 238-254, Springer Verlag.

[DJL84] P. Deransart, M. Jourdan and B. Lorho, Speeding up Circularity Tests for Attribute Grammars, *Acta Inf. 21*, (Dec. 1984), 375-391.

[DJL88] P. Deransart, M. Jourdan and B. Lorho, Attribute Grammars - Definitions, Systems and Bibliography, *LNCS 323*, (1988), , Springer Verlag.

[GrE90] J. Grosch and H. Emmelmann, A Tool Box for Compiler Construction, *LNCS 477*, (Oct. 1990), 106-116, Springer Verlag.

[Gro92] J. Grosch, Transformation of Attributed Trees Using Pattern Matching, *LNCS 641*, (Oct. 1992), 1-15, Springer Verlag.

[Gro] J. Grosch, Ag - An Attribute Evaluator Generator, Cocktail Document No. 16, CoCoLab Germany.

[IRL92] IRL, *Industrial Robot Language, DIN 66312*, Beuth-Verlag, Berlin, to appear, 1992.

[JOR75a] M. Jazayeri, W. F. Ogden and W. C. Rounds, On the Complexity of Circularity Tests for Attribute Grammars, *ACM Symp. on Prin. of Programming Languages 2*, (Jan. 1975), 119-129.

[JOR75b] M. Jazayeri, W. F. Ogden and W. C. Rounds, The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars, *Comm. ACM 18*, (Dec. 1975), 679-706.

[JoP88] M. Jourdan and D. Parigot, More on Speeding up Circularity Tests for Attribute Grammars, rapport RR-828, INRIA, Rocquencourt, Apr. 1988.

[Kas80] U. Kastens, Ordered Attribute Grammars, *Acta Inf. 13*, 3 (1980), 229-256.

[Kie91] K. Kiessling, Entwurf und Implementierung des Frontends eines Modula-3-Übersetzers, Diplomarbeit, GMD Forschungsstelle an der Universität

Karlsruhe, Apr. 1991.

[Knu68] D. E. Knuth, Semantics of Context-Free Languages, *Mathematical Systems Theory 2*, 2 (June 1968), 127-146.

[Mar90] M. Martin, Entwurf und Implementierung eines Übersetzers von Modula-2 nach C, Diplomarbeit, GMD Forschungsstelle an der Universität Karlsruhe, Feb. 1990.

[RaS82] K. Räihä and M. Saarinen, Testing Attribute Grammars for Circularity, *Acta Inf. 17*, (1982), 185-192.

[VSK89] H. H. Vogt, S. D. Swierstra and M. F. Kuiper, Higher Order Attribute Grammars, *SIGPLAN Notices 24*, 7 (July 1989), 131-145.

[Vog93] H. H. Vogt, *Higher Order Attribute Grammars*, PhD Thesis, University of Utrecht, Feb. 1993.

Contents