PROSPECTRA
an ESPRIT Project

Attribute (Re)evaluation in OPTRAN

Peter Lipps, Ulrich Möncke, Matthias Olk, Reinhard Wilhelm

Universität des Saarlandes
6600 Saarbrücken
Bundesrepublik Deutschland

Deliverable Item S.1.3 - R - 5.0

1987 - 01 - 29

Distribution: public

ABSTRACT

A transformation of a tree decorated according to some attribute grammar may leave the tree containing attribute inconsistencies. An attribute reevaluation algorithm computes new attribute values for affected attribute instances. It has to guarantee, that never an inconsistent attribute value is accessed. Reps' algorithm performs this task in time $O(|\text{affected region}|)$. It is *data driven* as changed values trigger recomputations of attribute instances dependent on them. After each transformation, a complete update of the effected instances is performed. Reps' algorithm is compared with the data driven reevaluation scheme used in OPTRAN. It uses the same strategic information in the initial attribute evaluation and the reevaluation process. Furthermore, we present a *demand driven* scheme for attribute reevaluation. It does not have the linear time complexity for each update after one transformation but, depending on the situation, often compares favourably with the data driven scheme for series of transformations. In addition, the linear time complexity of the data driven reevaluation algorithm needs fast convergence using an equality test between old and new attribute values. It is thus necessary, to keep the attribute values at (almost) all instances. The demand driven reevaluator does not need all the

old attribute values. It can flexibly trade time for space. We also describe the handling of space consuming attributes, e.g. tables, lists and trees, in the reevaluation algorithm. An integrated version of data driven and demand driven reevaluation using these features has been implemented in the OPTRAN system.

public

Peter Lipps, Ulrich Möncke, Matthias Olk, Reinhard Wilhelm
Universität des Saarlandes

in the Project

PRO**gram** development by
SPECification and
TRAnsformation

sponsored by the

Commission of the European Communities

under the

European
Strategic
Programme for
Research and development in
Information
Technology

Project Ref. No. 390

Attribute (Re)evaluation in OPTRAN

Peter Lipps, Ulrich Möncke, Matthias Olk, Reinhard Wilhelm

FB 10 - Informatik
Universität des Saarlandes
D - 6600 Saarbrücken
Federal Republic of Germany

1. Introduction

Most of the literature on attribute grammars concerns attribute evaluation strategies for "static" trees, i.e. trees which are constructed by a parser, decorated with attribute values according to a given attribute grammar, and then passed on to code generation. Few systems, such as the Cornell program synthesizer generator [Re82] and the OPTRAN system [GPSW86] work with dynamically changing trees. While the Cornell program synthesizer generator supports subtree replacement upon user request, the OPTRAN system supports general transformations on attributed trees whose applicability may be restricted by predicates on attributes. OPTRAN is therefore well suited for batch type transformations, e.g. code optimization, source-to-source translation, code generation, etc. The existence of special strategies [Kr74, We83] (e.g. bottom-up-left-right, top-down-left-right) makes it possible to perform a series of transformations in batch mode. An interactive mode is also possible by programming user interaction into predicates. An OPTRAN transformation rule has the form

transform <tree template>
  if <predicate on attributes of the input tree template>
  into <tree template>

Application of a rule thus requires a syntactic match, i.e. finding an instance of the input template in the subject tree, and a successful evaluation of the applicability predicate on attribute instances of the matched region of the tree, cf. figure 1.

Application of a transformation rule, in OPTRAN - as well as in Cornell program synthesizer generator-generated transformers, may destroy the consistency of attribute values in the transformed region. Due to long reaching attribute dependencies, attribute instances far away from the point of change may be affected.

It is the job of an attribute reevaluation algorithm to restore (partial) consistency of attribute values in the tree. It must be guaranteed, that never an inconsistent value of an attribute instance is accessed. The Cornell program synthesizer generator offers a data driven reevaluation scheme. After each

Research reported herein was partially supported by the Deutsche Forschungsgemeinschaft, project "manipulation of attributed trees" and ESPRIT, project PROSPECTRA.

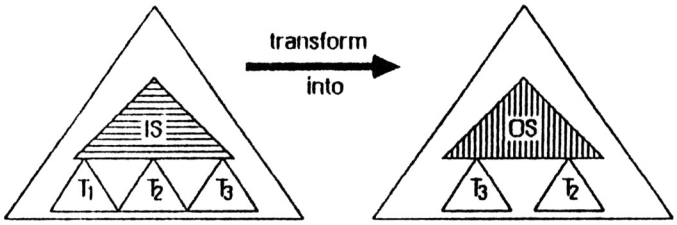

Figure 1
IS: area "covered" covered by the input template
OS: area changed according to the output template, the "transformed area"
T2,T3: rearranged subtrees

transformation, a reevaluation process is started at the point of change. It uses the attribute dependency graph to locate attribute instances, which may have been affected by updated attributes. The reevaluation algorithm works in time $O(|N|)$, where N is the number of affected attributes. It is thus hard to beat, as long as the effort for a complete update after one transformation is regarded. However, it may happen, that subsequent transformations lead to several updates of the same set of attribute instances, although none of them was needed in between.

The OPTRAN system offers the user a choice to specify data driven or demand driven reevaluation for subsets of attributes. Data driven reevaluation works similar to Reps' algorithm; differences are described in section 3. In particular, the OPTRAN reevaluator makes more use of generation time information. The demand driven algorithm is able to delay the reevaluation of attributes, until their values are needed. A labeling algorithm signals the change to all attributes, which depend on changed attributes. The first labeling effort after a transformation may be large; subsequent labeling runs, however, may lead to rapid convergence, as the labeling process terminates at appropriately labeled attributes. The demand driven algorithm, thus allows several transformations in a row updating attributes only as needed by some consumer, e.g. an applicability condition of a transformation rule or the display manager of an interactive program transformation system.

Demand driven reevaluation has a clear advantage over data driven reevaluation, if

- the attribute evaluation rules and the equality test on attribute values are much more expensive than the (cheap) labeling actions, e.g. for nonscalar attributes;
- the environment is space critical, which often is the case; demand driven reevaluation, in principle, does not need storage for attribute values;
- a series of transformations has long reaching effects on attribute instances, which themselves are not needed by any of the transformations in the series. Premature reevaluation may even lead to the signaling of violated context

conditions, e.g. type inconsistency or use of uninitialized variables after the introduction of new variables in the declaration part. For that reason, demand driven reevaluation was envisioned in [Re82] (now implemented in the Cornell Synthesizer).

Section 2 introduces the necessary terminology. Section 3 describes the data driven reevaluation scheme implemented in OPTRAN, section 4 the demand driven scheme. In section 5, we present a way to trade time for space in demand driven reevaluation. Experiences are reported in section 7.

2. Attribute values and consistency

We assume the familiarity of the reader with the usual terminology about attribute grammars, c.f. [Co84, DJL86] .

At every production, we call the occurrences of inherited attributes at the father and the synthesized attributes at the sons *used attribute occurrences* and the occurrences of synthesized attributes at the father and of inherited attributes at the sons *defined attribute occurrences.* We require the existence of exactly one semantic rule

$$(a,i) = f((b_1,i_1), \ldots, (b_k,i_k))$$

for each defined occurrence *(a,i)* in a production $p: X_0 \rightarrow X_1 \cdots X_n$. Following [Co84] and [BG86] , we regard *f* as a term built from function symbols from some signature. The system of equations of the form

$$(a,m.i) = f((b_1,m.i_1), \ldots, (b_k,m.i_k))$$

for a tree *t*, where *(a,m.i)* is the attribute instance of *(a,i)* at node *m.i*† , can be solved using one of the many available attribute evaluators in two ways: by interpreting the function symbols as functions first, or by formally solving the system in some term algebra and then interpreting the resulting terms.

Example:

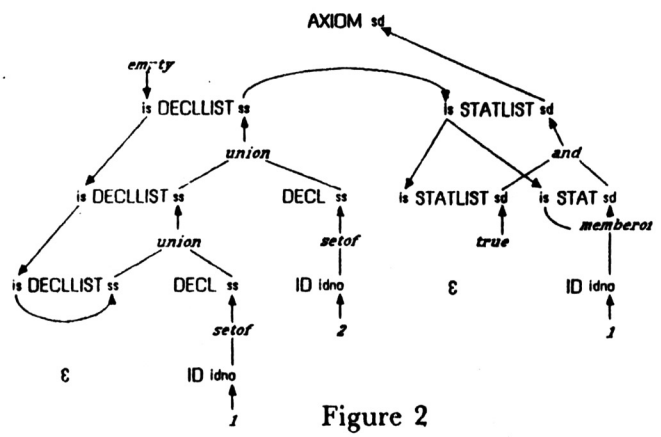

Figure 2

Figure 2 shows an attributed tree for a small program, consisting of a declaration

† m.0 stands for m

and statement part. The corresponding grammar is given in Appendix A. Each defined occurrence of an identifier is inserted into a set of identifiers (see attributes *ss* "*s*ynthesized *s*et" and *is* "*i*nherited *s*et"). For each applied occurrence it is checked, if a corresponding defined occurrence exists (see *sd* "*s*ynthesized *d*efined"). If all applied identifiers are declared the value of sd^{AXIOM} is true, otherwise false.

The function symbols in this grammar are *union, setof, memberof, empty, and, true, 1, ...* . The value of sd^{AXIOM} is true; if formally solved, it is the term

and(memberof(1, union(union(empty, setof(1)), setof(2))), true).

For any tree *t* of a noncircular attribute grammar and any instance *(a,m.i)* of an attribute *a* we have the following close correspondence between its formal value *T(t,a,m.i)* and the part *D(t,a,m.i)* of the dependency graph *D(t)* directed towards *(a,m)*. The formal value of *(a,m.i)* is the term

$$T(t,a,m.i) = f(T(t,b_1,m.i_1), \cdots ,T(t,b_k,m.i_k))$$

Thus, *T(t,a,m.i)* taken as an ordered, labeled tree, and *D(t,a,m.i)* with an added order on the entering edges, are closely related to each other; removing the function symbols from *T(t,a,m.i)* and introducing nodes representing attribute instances yields *D(t,a,m.i)*.

A value of an attribute instance *(a,m)* is said to be *consistent,* iff it agrees with the interpretation of *T(t,a,m)*.

3. Data driven reevaluation in OPTRAN

A data driven reevaluation scheme is defined by: A new consistent attribute value for an attribute instance is computed if at least one of the arguments of the function defining the attribute's value has received a new value, and this update action is executed, as soon as all its arguments have either new consistent values or old values, guaranteed to have remained consistent.

The reevaluation process, described by Reps, works on an attribute dependency graph ("model"). The starting graph contains the attribute dependency graph which is local to the instance of the output template. It is expanded by the superior characteristic graph at the root and the subordinate characteristic graphs at the leafs of the output template. Characteristic graphs contain edges between attribut instances if there is a path in D(t) leading from one instance to the other. Topological sorting of the nodes in the model graph is interleaved with expansion of the graph: An node representing an attribute instance is ready for evaluation if all the arguments of the defining semantic functions are evaluated. Since nodes representing computed attribute instances are removed from the model, a node is ready for computation iff it has indegree 0. If an attribute has changed its value and influences attribute instances in a neighbour production, the model is expanded: The characteristic graph located at the frontier to the neighbour production is removed from the model and the production local graph is added together with the characteristic graphs at the outside positions of the neighbour production

excluded the point of expansion itself. The expansion stops if no more attribute instances outside the model graph are affected by changed instances.

In [Mö85] , a reevaluation algorithm is introduced for the purpose of restricting the area of reevaluation as in [Re82] . It uses labels N and E for attribute instances (N = value has changed, E = value has remained equal). Computations of updated attribute values and labeling of attributes are executed in an interleaved fashion, as soon as the arguments are computed or labeled.

The reevaluation algorithm is parameterized by the kind of strategic information driving it. The actual strategy parameters may be local attribute dependency graphs as in Reps' algorithm. They may also be evaluation plans as generated for initial attribute evaluation, cf. section 4 and [KW76] . This strategic information indicates, which attribute instances are candidates for the next update actions.

Productions in a tree being afflicted by the current state of reevaluation are called *active*. Those not yet afflicted are called *inactive*. A production becomes active, if at least one of its used attributes is labeled N. This means the production belongs to the area, where the reevaluator has to work.

All productions in the area changed by the transformation are initially active. Inside that area, it is necessary to reevaluate, because there exist no old attribute values except at the border.

The actions of the reevaluation algorithm are now depicted by the corresponding situations in the tree.

3.1. Actions of the labeling algorithm outside the transformed area:

(1) The scheduler using strategic information demands the evaluation of an attribute instance *a*. There are two cases:

(a) The arguments of the attribute instance *a* are all labeled E. Their values are not changed by transformation. Therefore the value of instance *a* cannot be changed either. Hence, instance *a* is labeled by E. This is a so called *inner*† *E-propagation*, c.f. figure 3.

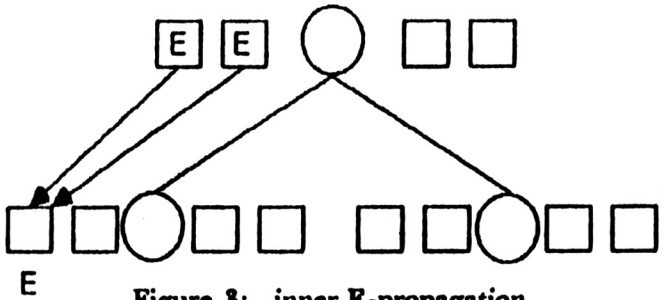

Figure 3: ***inner* E-propagation.**

Note, that the old labels are inside the attribute boxes, and the new label is outside the attribute box.

(b) At least one of the arguments is labeled N, i.e. its value has changed.

† inside the production

To find out, whether the label of *a* has to be changed, too, *a* has to be recomputed. New and old value are compared. If both are equal, instance *a* is labeled **E**, otherwise **N**, and a neighbour production becomes active. This is called a *compute-and-label-step*, cf. figure 4.

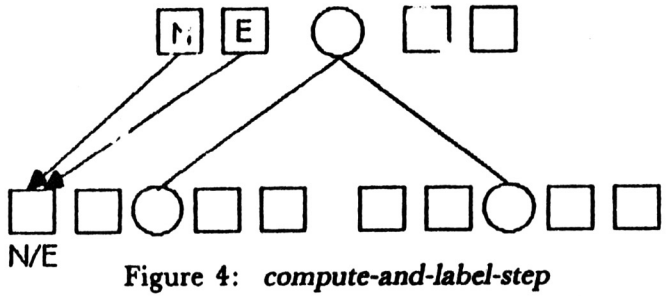

Figure 4: *compute-and-label-step*

(2) The strategic information requires the evaluator to switch to the neighbour production of the i'th son or the father:

(a) Reevaluation is necessary, if the neighbour production is active, i.e. at least one attribute instance is labeled **N**. The evaluator continues with the strategic information of the new production, cf. figure 5.

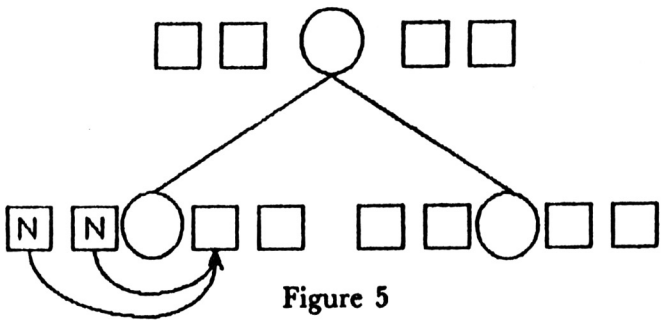

Figure 5

(b) Reevaluation is not necessary, if the neighbour production is inactive, i.e. no attribute instance is labeled **N**. All that has to be done is an *outer*† **E**-*propagation*. All used attributes at a node on the border of the production depending only on already **E**-labeled attributes are labeled **E**. The evaluator continues according to the strategic information of the actual production, cf. figure 6.

† outside the production

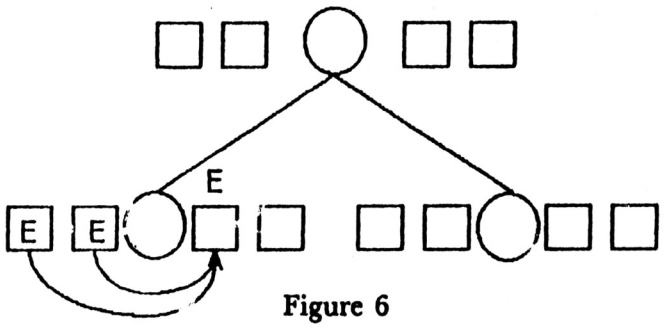

Figure 6

(3) Upon first activation of an instance of a production, all its attribute instances guaranteed not to have changed their value, must be labeled E, cf. figure 7.

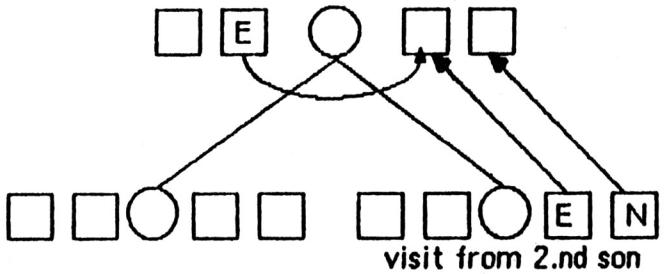

Figure 7

3.2. Actions of the reevaluation algorithm inside the transformed area

(1) All attribute computations as indicated by the actual strategic information have to be performed. If the attribute instance lies on the border of the output template, new and old values are compared and the instance is appropriately labeled. If the label is **N** , the neighbour production becomes active.

(2) All production instances, which make up the transformed area, must be visited for a complete reevaluation.

Using labels and characteristic graphs at any point and any time of reevaluation makes it possible to control the choice of the active production:

A visit of a neighbour production is *productive* iff

- at least one used attribute instance would become available, i.e. labeled with **E** or **N**, after return from the visit
- and all defining attributes the used attribute depends on are reevaluated and at least one is labeled with **N**.

A last visit to the active neighbour may be scheduled for propagating the changes to instances on which no used instance depends. Scheduling visits with respect to productivity decreases the tree walking effort necessary for reevaluation. This check of productivity is only possible, if the status of all defining attribute instances is available. The labeling scheme described above achieves this improvement. In contrast to Reps' algorithm [Re82] (chapter 5), the information about transitive dependencies (represented in the characteristic graphs) is used all the time and at any time for checking the productivity.

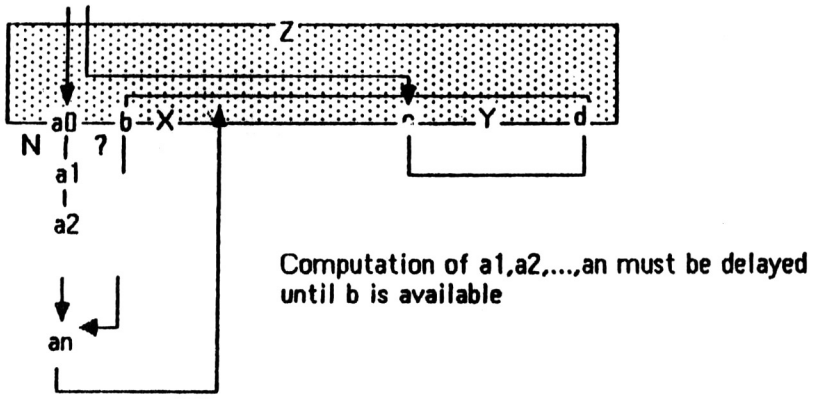

Figure 8

A visit to the subtree at X is scheduled iff both a_0 and b are available. Without this criterion an evaluator could follow the bold line until further arguments are needed (here: attributes depending on b) to compute the value of a_n.

4. Reevaluation using plans

Reevaluation using characteristic graphs (local dependency graphs) in general leaves a nondeterministic choice where to continue, i.e. which instance to evaluate next and/or which active neighbour production to visit. Reps' algorithm eliminates this nondeterminism by inserting evaluable instances into a topologically sorted worklist. Another algorithm proposed by Reps [Re82] (chapter 9.2) uses the automata described by [KW76] .

Any evaluation order can be induced by an order precomputed at generation time. The attribute instances of the whole syntax tree may be grouped together in classes of simultaneously evaluable instances, and these classes may be totally ordered. This mapping of the attribute dependency graph can be realized with the help of precomputed information. Both grouping and ordering has to be done with respect to the attribute dependencies expressed in the graph D(t). Once such an ordered partition of all instances is installed, any evaluator schedules the attribute instances according to this order. The evaluators play a "pebbling game" with values and labels (E, N) on the ordered partition of attribute instances. In the tree evaluation method of [KW76] the ordering of instances is implied from the dynamic behaviour of the evaluation automata. In our approach, the order is made explicit. Classes of instances are scheduled as a whole. Of course, after a transformation the ordered partition must be adjusted. The restriction of the ordered partition to attribute instances at a tree node shows the visits an evaluator will make to the upper tree fragment rsp. the subtrees at this node. The restriction to a production instance shows the sequence of computations for classes of defining attributes and visits to neighbour productions, which will return with classes of used attributes. After a transformation, the process of evaluation starts from the output region, i. e. at the production instances of the output template and activates the neighbour productions, if there is a change in one of the defining attribute classes.

The OPTRAN reevaluator uses precomputed evaluation plans to direct both initial evaluation and reevaluation. Plans are straight line programs. There is no branching in a plan (in contrast to the [KW76] evaluator). A pointer into a plan is all that is necessary to find the position where to continue. Besides ordering the attribute occurrences of a production, a plan groups all those attribute occurrences, which can be (re)evaluated together. This may decrease the tree traveling effort. In 4.1, we describe reevaluation using plans, and in 4.2, we sketch, how plans are generated in OPTRAN.

4.1. Plans at work

An (attribute evaluation) plan for a production p: $X_0 \rightarrow X_1 \cdots X_n$ is a sequence of *evaluate-class-of-attributes-* and *visit-i-th-neighbour-* instructions, where *i* may be 0 for a visit to the father or may be in [1..n] for a visit to a son. For i = 0 there are subsequences of the form:

compute-class-of-synthesized-attributes-of-the-father ; visit-father.

For $1 \leq i \leq n$, there are subsequences of the form:

compute-class-of-inherited-attributes-of-the-i-th-son ; visit-i-th-son.

The plans are generated in a way,

- that no visit (except maybe the last one) to a neighbour is made without the guarantee, that upon return new values of used attribute instances will have become available; at least one used attribute will be labeled with E or N after returning from the visit. In this way, the productivity criterion is incorporated in the plans. For example, the attributes a_0 and *b* (fig. 8) are contained in one class,
- that all the attribute instances needed to evaluate a class of defined instances in a plan precede that class in the plan.

The use of plans and their advantages, i.e. space and time efficiency, have long been known [KW76, DJL86] . How are they used in attribute reevaluation? The main question is, how does the reevaluator find the right start in the plan, when a production is visited for the first time? Let us distinguish the *visitor* and the *visitee*. The executed visit in the visitor's plan indicates which class of instances it evaluated. Hence, the starting point in the visitee's plan is the point right after the last visit to the visitor. If there is no previous visit, the starting point depends on the entry position, i.e. the attribute class, which the visitor has defined at last for use of the visitee. A visit to a neighbour production in a plan may be skipped, if all earlier attribute instances at that neighbour in the plan are E-labeled. These attribute instances are members of the classes which the visitor has already defined for the neighbour. An outer E-propagation is performed instead.

Sometimes skipping of a visit would be possible, even though not all these instances are E-labeled, but this cannot be recognized by the evaluator.

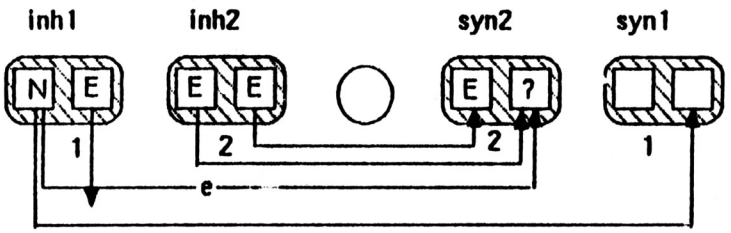

Figure 9

Let inh_i be the last class defined for the neighbour and syn_i be the class, which would be available after the visit. Suppose the attribute instances in inh_i are labeled E and at least one of the attribute instances in inh_{i-1} is labeled N. Then the visit must be scheduled due to the worst case assumption infered by the total order on classes, that some attribute in inh_j ($j \leq i$) may affect an attribute instance in syn_i. Of course, in the actual syntax tree, there need not be such a dependency. Figure 8 shows this situation: If there was no path *e* from the inherited attribute in class inh_1 to the attribute in class syn_2 visit 2 could be skipped.

There may be several plans for one production. For each instance of a production, the right plan is selected by taking the upper and lower context into account. The lower context is given by the subordinate characteristic graphs at the sons. They can be determined by some bottom-up tree automaton constructed at generation time. The upper context is given by the ordered partition of the attribute instances at the father node. An ordered partition [Ni83] is a sequence $((inh_1,syn_1), \ldots ,(inh_k,syn_k))$ of pairs of classes, where inh_i (syn_i) are subsets of the inherited (synthesized) attributes of a nonterminal. The set inh_i comprises all the inherited attributes of the nonterminal, which can be evaluated at the next visit, if the values of all the attributes in the sets $syn_1, \ldots ,syn_{i-1}$ are known. The set syn_i consists of all the synthesized attributes, which can be evaluated, when the values of all the attributes in the sets $inh_1, \ldots ,inh_{i-1}$ are known. Such an ordered partition represents one of the evaluation orders for the attributes of that nonterminal, c.f. section 4.2. The appropriate ordered partitions for the sons of the production instance are computed by a top-down tree automaton using the ordered partition of the father, the subordinate characteristic graphs at the sons and the local dependency graph of the production.

As a byproduct, an ordering of all classes, which belong to this production, is computed. The evaluation plan refers to this ordering. Once a plan is selected, the starting position depends on the entry position.

4.2. Construction of plans in OPTRAN

Evaluation plans are produced at generation time using grammar flow analysis [Mö85, Mö86] . The computation of evaluation plans is based on the preceding computation of the subordinate characteristic graphs [Re82, DJL86] and the ordered partitions. Subordinate characteristic graphs are computed on the

bottom-up grammar-graph, containing two sorts of nodes, *nonterminal nodes* and *production nodes.*

Edges leading from nonterminal nodes to production nodes represent the right side of productions. Associated with them is a function mapping the grammar flow information at incoming edges. In the case of subordinate characteristic graphs, the function associated with a production

- takes any combination of subordinate graphs already available at the sons,
- composes it with the production local dependency graph, and
- restricts the resulting graph to the attributes of the left side.

Edges leading from production nodes to nonterminal nodes combine the grammar flow information contributed by the different productions for that nonterminal. In the case of the subordinate characteristic graphs, the sets of characteristic graphs for the different productions of one nonterminal are united, and the union is the new information for that nonterminal.

This computation of sets of graphs is continued until a fixpoint is reached.

The computation of ordered partitions and plans is done by a top-down grammar flow analysis, i.e. iteratively on the *top-down grammar-graph.* This graph is obtained from the bottom-up grammar-graph reversing the edges' direction.

Together with the computation of subordinate characteristic graphs a bottom-up tree automaton is generated which at runtime, i.e. attribute evaluation time, computes the actual subordinate characteristic graphs for the individual tree. With the subordinate characteristic graphs available, the ordered partitions and a top-down tree automaton determining the individual ones at run time is generated. This computation is somewhat complicated and therefore described in more detail.

One step in this top-down process works as follows: It uses the ordered partition computed previously at the left side of the production, the local dependency graph of the production, and the characteristic graphs stemming from the preceding bottom-up analysis. An ordered partition at the father node indicates the order of attribute evaluation at instances of that node. The step under consideration has to determine this order for each son. The generator now takes in turn any inherited class in the ordered partition at the father and finds for each son the class of now computable inherited attributes. Using the subordinate characteristic graphs the class of synthesized attributes then computable is evident. In this way, several possible sequences of pairs (inh_i, syn_i) can be computed, each indicating one possible evaluation order for the attributes of that nonterminal in that context. One is then selected by some heuristics, i.e. left-to-right preference, productivity of an inherited class or such.

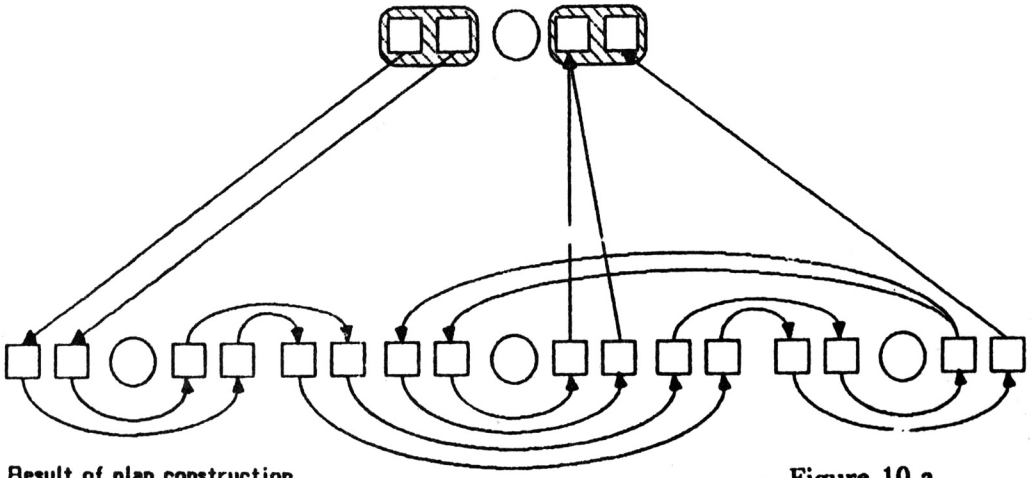

Figure 10.a

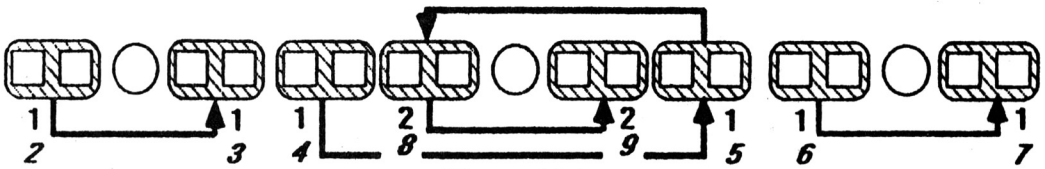

Figure 10.b

The evaluation plans for a production are straight line programs determining attribute evaluation and visits to neighbouring productions according to the total order given by the ordered partitions. They can be computed together with the ordered partitions at generation time. At run time, a top down tree automaton selects the appropriate plan for each production from its state at the father and the states (at the sons) of the bottom-up automaton computing the actual subordinate characteristic graphs. Our generating tools permit the precomputation of characteristic graphs [Kn68] as well as approximative graphs (e.g. the IO-graphs [KW76]).

Considering the relationship between nonterminals and subordinate characteristic graphs, and the relationship between subordinate characteristic graphs and ordered partitions some particular cases may be distinguished. This distinction is not only of theoretical interest for classification for attributed grammars but leads to practical consequences in the design of the transformation runtime system.

Let's start with the most general case and then stepwise refine it to particular cases.

A set of characteristic graphs, computed for a nonterminal X induces a partition in equivalence classes of subtrees produced by X (figure 12 left side). Two subtrees are said to be equivalent if the characteristic graphs assigned to their roots are identical. Each class is represented by a characteristic graph. The set of subtrees, produced by X is in general infinite, while the set of classes is finite. Taking a subtree produced by X out of one class characterized by C, and combining it with a upper context, the ordered partition for X is determined. Formally seen, the ordered partition is a function of a characteristic graphs and upper tree contexts

(c.f. figure 11).

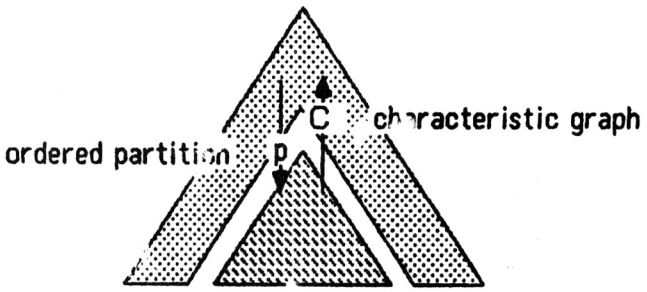

Figure 11

The upper contexts (tree fragments with the subtree removed) may be split into equivalence classes, too, each of them represented by an ordered partition (c.f. figure 12). Therefore, there exists a non-trivial set of ordered partitions for each characteristic graph, i.e. a one-to-many relationship between graphs and partitions.

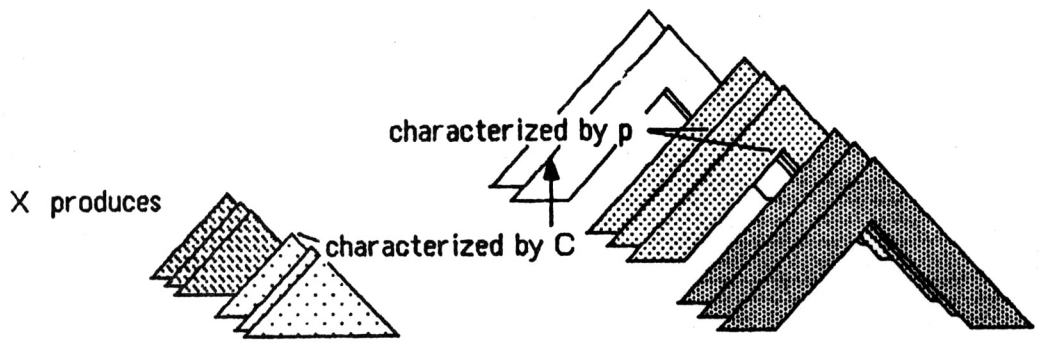

Figure 12

Now, let us discuss particular cases (Figure 13):

For each nonterminal of the grammar there may be exactly one characteristic graph, characterizing all subtrees produced by X. Then, there is no need for a bottom-up automaton for propagating graphs, because of the one-to-one relationship between nonterminals and graphs. This is the most desirable situation.

Using the approximative characteristic graphs (IO-graphs [KW76]) – assumed the grammar is absolutely noncircular – we will guarantee this situation of having one graph per nonterminal (by construction).

In addition to the above mentioned one-to-one relationship between nonterminals and graphs, there may be only one ordered partition for each of the characteristic graphs. Then, the top-down automaton is also not neccessary. In this case, the attributed grammar is l-ordered. (cf. [Ka80]). There is a one-to-one relationship between nonterminals and both characteristic graphs and partitions.

Of course, the latter is a refinement of the slightly more general case, where there is a one-to-one relationship between characteristic graphs and ordered partitions,

even if there is no one-to-one relationship between nonterminals and characteristic graphs. We call such a the grammar l-ordered with respect to characteristic graphs.

nonterminals → sets of characteristic graphs	characteristic graphs → sets of ordered partitions: one-to-one	one-to-many
one-to-one	no bottom up propagation no top down propagation	no bottom up propagation top down propagation
one-to-many	bottom up propagation no top down propagation	bottom up propagation top down propagation

Figure 13

It should be pointed out, that the heuristics for constructing the ordered partitions will not always succeed in constructing a one-to-one relationship between graphs and partitions, even if there is one.

Reevaluation using plans does not depend on the way they are generated. In particular, each simple-multi-pass partition [Al81, EF82] of attributes induces a totally ordered partition of attribute instances for each tree.

A promising approach is the mixture of a simple-pass partition method and the generation method described above.

Sometimes, use of such a precomputed partition is crucial. In analogy to the worst-case behaviour of the computation of characteristic graphs the number of ordered partitions may grow very fast. Unfortunately, this is not only a matter of theoretical interest. The large numbers of ordered partition can be observed in practical examples. The reason is, that the heuristics constructing ordered partitions tries to schedule attributes for computation as early as possible. Of course, for each attribute the time of scheduling depends on the upper tree context. Different contextsmay schedule the same attributes for earlier or later visits. Even if the different schedules affect only few attributes, the combination of them leads to an explosion in the number of partitions. From this viewpoint we would prefer to delay the computation of attributes, and in this way construct a smaller set of coarser ordered partitions. On the other hand, practical experience shows, that in general the attribute grammars are "almost" pass oriented (c.f. figure 14), i.e. with the exception of some "runaway" attributes, the remanining set may be scheduled in simple passes. Then, attribute evaluation is seen as a sequence of evaluation phases, where some phases are passes (w.l.o.g. left-to-right, right-to-left) and some phases are not pass computable. Therefore, we have parametrized our ordered partition construction by the pass partition. It must schedule attributes according to the precomputed pass partition, i.e. in particular delay some evaluations for all attributes which are computable in a pass. Therefore, most of the time visits are

performed in passes. Only for those attributes, which do not fit into the pass scheme, the heuristics is free in planning their visits. Only sets of such attributes may be splitted for different visits at one nonterminal (c.f. figure 14).

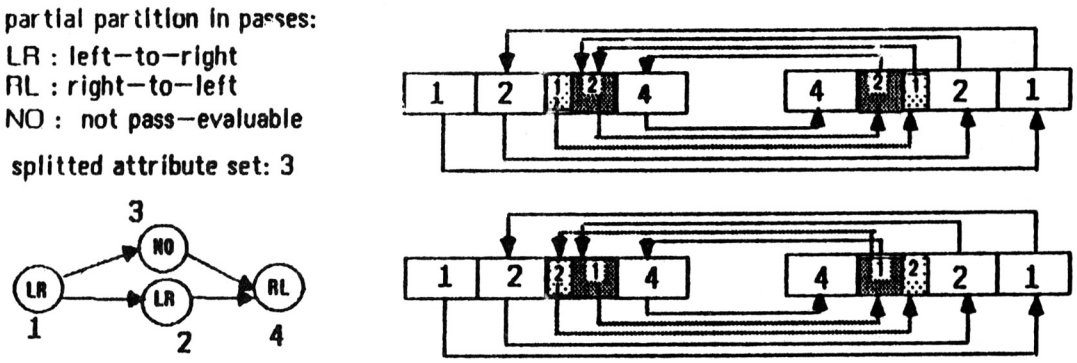

Figure 14

Of course, if the whole attributed grammar is pass computable, no work is left for the ordered partition constructor.

This method prevents the combinatoric explosion, mentioned before (e.g. reduces the number of ordered partitions from 50 to 3 in a Pascal-grammar which is attributed for code generation). The space needed both for propagating automata and for production local attribute evaluation plans can be drastically reduced, without restricting the power of the attribute evaluator.

5. Demand driven reevaluation

We now describe a scheme for demand driven evaluation. The principle is, to delay the reevaluation of attribute instances, until there is a demand for their values, i.e. an attempt to access them.

The straightforward recursive evaluator is the applicative evaluator P4 in [En84] . It does not keep any attribute values. When a value is needed, P4 recursively walks through the (noncircular) dependency graph, until it returns with the value. Its drawback is the frequent reevaluation of the same (namely shared) attribute instances, even in the case of nontransformed trees. Its worst case complexity is exponential in the size of the attributed tree.

We will go to the other extreme in this section, assuming that the recursive evaluator will leave the (new) value at each instance (much like P5 in [En84]), when returning. It must then be guaranteed, that no inconsistent attribute values will ever be accessed. For that purpose, it has to be signaled to all (possibly) affected attribute instances, that their value may have been changed due to a transformation.

We introduce two labels to distinguish (possibly) inconsistent attributes from (certainly) consistent ones, an I-label (for inconsistency) and a K-label (for

consistency). Demand driven reevaluation is split into two phases, a *labeling phase* distributing I and K-labels and an *updating phase*, recursively recomputing values of needed attribute instances. A labeling phase is executed after each transformation, an updating phase before each transformation, which issues a demand. Each updating phase relabels the updated instances as consistent (label K).

If a transformation involves demand attribute instances, those instances and all instances depending on them (instances of the so called "depending area") will be I-labeled. For the moment we will state that an attribute instance is involved in a transformation, iff it's value may have changed as a consequence of this transformation. (One may ask for a stronger requirement at this point: 'the value certainly must have changed").

Let's sum up the meaning of these new labels:

- An attribute instance *(a,m)* is I-labeled, if a preceding transformation may have changed the term *T(t,a,m)* and if there was no intervening reevaluation so far.
- An attribute instance *(a,m)* is K-labeled, if the interpretation of the term *T(t,a,m)* certainly equals the last value computed for it and stored at the instance.

Note, that the term *T(t,a,m)* is not interpreted and checked for equality during the labeling phase.

Between any labeling and updating phases the following invariants hold:

Invariant 1:

All instances depending on an I-labeled instance are also I-labeled .

Invariant 2:

If an instance is K-labeled, none of the instances it depends on is labeled I.

Invariant 2 is a consequence of invariant 1. However, both invariants nicely mirror two different aspects of demand driven reevaluation. Invariant 1 is significant for the labeling phase. It means, that all instances, which transitively depend on an I-labeled instance, have to be labeled I. It also supplies one convergence criterion for the labeling phase: stop at every I-labeled instance.

Invariant 2 is significant for the updating phase. It means, that the updating phase should not continue over any K-labeled instance. In particular has every instance a consistent value, all of whose arguments are K-labeled.

If the value of an I-labeled attribute instance has to be evaluated, the recursive evaluator is called. We will give an example of a modified recursive evaluator in section 6.

5.1. Footholds in demand driven reevaluation

As described in the previous section, pure demand driven attribute evaluation does not need storage of attribute values. Any time an attribute value is needed, the recursive evaluator is called to (re)compute it. The completely opposite approach, which stores the value with each instance, was used in the last section. In FNC [Jo84] only the values of synthesized attributes are stored, inherited attributes have to be recomputed, but the saving in space should not be substantial.

We describe a compromise, that allows to flexibly trade time for space. The user specifies some attribute occurrences as *footholds*. Any instance of a foothold will have its last computed value stored with it, all other instances of *complex* attributes will have no permanent value. Of course, the introduction of footholds will not change the worst-case time complexity, which is exponential in the tree size, but in practice declaring the right occurrences (e.g. symbol table attribute at the top node of the statement part) as footholds decreases the runtime significantly.

6. Integration of data driven and demand driven reevaluation

This section describes a combination of data driven and demand driven reevaluation. We assume (as is the case in OPTRAN), that a subset of the attributes is specified as to have non-atomic domain. We call them *complex* attributes. Each instance of a complex attribute is evaluated by demand. Footholds for them may be specified. Only for these footholds the values will be kept. Attribute instances with atomic domain not depending (even transitively) on any complex attribute instance are called *regular*. They are evaluated and reevaluated in a data driven fashion. Their values are stored. Atomic attribute instances depending on complex attribute instances will also be reevaluated by demand, but their values will be kept. These attribute instances and the complex ones together constitute the class of so called *demand* attributes. Note, that there may be both regular and demand instances for the same attribute occurrence in a production's instance.
Example:

> If *is/ss*, the set of defined identifiers in figure 2, is declared to be a complex attribute, then sd^{AXIOM} will be evaluated by demand since it transitively depends on a complex attribute instance. Deleting the subtree at STATLIST will cause sd^{AXIOM} to become a regular instance with value true.

The reevaluation algorithm of section 5 is now extended, as to serve for the reevaluation of both regular and demand attribute instances. For this reason, the above mentioned labels K (value is consistent) and I (value may be inconsistent) are introduced.

There are now two types of labels, N and E for regular attribute instances and K and I for demand attribute instances.

Reevaluation in the combined scheme starts with data driven reevaluation for the regular attribute instances and the labeling phase for the demand attribute instances. N and E are used during data driven reevaluation with their

conventional interpretation. When this is finished, their interpretation changes. Both N and E then signal, "consistent regular attribute instance".

K- and I-labels control demand driven evaluation. Note, that K and I labels are distributed without any recomputation, not even inside the output template, while E/N-labeling and reevaluation work in an interleaved way.

After termination of a labeling phase invariant 1 is established, i.e. all successors of an attribute instance labeled I are labeled I, too.

In the combined reevaluator, a production becomes active, if at least one used attribute instance is labeled N or newly labeled I. This means, the label has changed from K to I in the actual labeling phase. Any instance, labeled I by a previous reevaluation phase signals: no further action for the evaluator. Invariant 1 states, that all the successors of that instance are labeled I.

We now give an example of a recursive evaluator, considering demand attribute instances, K/I-labels and footholds for complex attribute instances. Note, that footholds are specified for complex attribute occurrences. Propagation of I-labels is part of the scheduler's work, which is not presented here.

```
procedure Eval (ai₀ );
  /* Eval tests label of ai₀ and calls recursive evaluator if necessary */
def
  - attribute instance to be computed:                ai₀
  - label of attribute instance ai:                   label_v (ai) ∈ {E,N,K,I}
  - foothold label of attribute instance ai:          label_f (ai) ∈ {No,Yes}
  - test, if attribute instance has complex domain:   Complex(ai)
begin
  case label_v(ai₀) of
    E,N   : null;
            /* a regular attribute instance has a consistent value */
    I     : DemandValue(ai₀);
            /* evaluate inconsistent demand attribute instance */
    K     : if Complex(ai₀ ) then
              if label_s(ai₀) = No then
                /* ai₀ is a complex attribute instance and
                   no foothold and therefore has to be reevaluated */
                DemandValue(ai₀);
              else
                /* ai₀ is a foothold with consistent value */
                null;
              fi;
            else
              /* ai₀ is a non-complex attribute instance
                 with consistent value */
              null;
            fi;
  esac;
endproc;
```

```
procedure DemandValue (ai0);
   /* recursive evaluator */
def
   - function, which computes the value of ai0:        f
   - arguments of function f                           ai1  (1≤i≤rank(f))
   - corresponding attribute occurrence:               Occurrence(ai)
   - test, if attribute occurrence ao is a foothold:   Foothold(ao)
begin
   for i: = 1 to rank(f) do
      /* evaluate arguments of function f */
      Eval (ai1);
   od;
   ai0: = f(ai1, ... ,airank(f));
   labelv(ai0): = K;
   /* ai0 now has a consistent value */
   if Foothold(Occurrence(ai0 )) then
      /* corresponding attribute occurrence is specified as foothold;
         label the attribute instance accordingly */
      labelf(ai0): = Yes;
   else
      labelf(ai0): = No;
   fi;
endproc;
```

6.1. Actions outside the transformed area

The combined evaluator starts in the transformed area of the tree. It performs data driven evaluation, as long as it does not encounter instances labeled I or K, i.e. demand instances. It behaves like the pure demand driven reevaluator of section 4 when reevaluating demand instances. The following subsections describe the situations involving both demand and regular instances.

No recomputation of an instance's value needs to be performed, if all arguments are labeled E or if at least one argument is labeled K or I. In the latter case, demand attribute instances are involved, whose updating is delayed.

However, if one argument is inconsistent, the attribute instance to be computed may be inconsistent. Therefore, the evaluator labels an attribute instance I, if at least one of its arguments is labeled I.

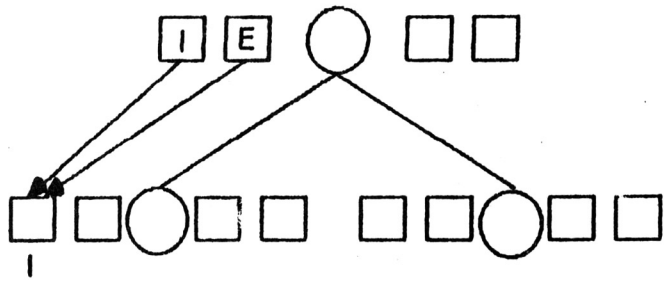

Figure 15: *inner* **I-*propagation***

The value of an attribute instance may have changed, if it has as arguments a consistent demand attribute and a regular attribute, whose value has changed. Therefore, the evaluator must label the goal attribute I, if no argument is labeled I, but at least one K and one N.

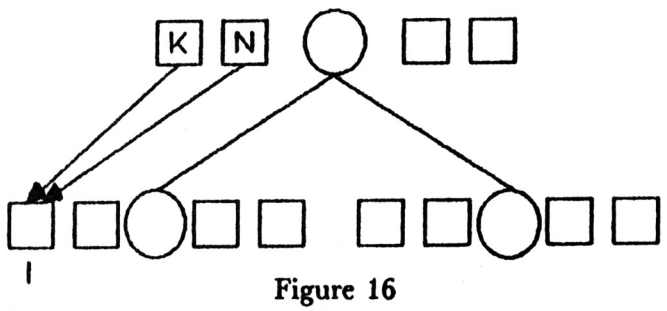

Figure 16

An attribute instance has a consistent value, if it depends on consistent demand attributes and on regular attributes, the values of which have not changed. An instance may have been inconsistent before, so that it keeps its inconsistency. That means, no I labels are removed during the labeling phase.

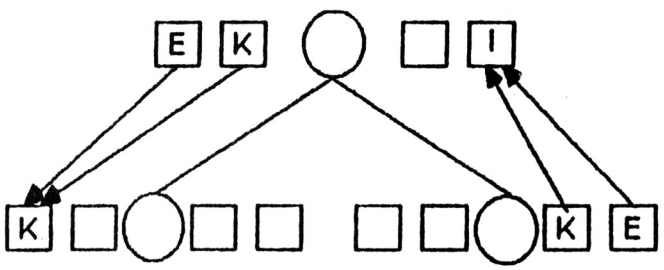

Figure 17

6.2. Actions inside the transformed area

Inside the transformed area all attribute instances are labeled I, which have at least one argument labeled K or I. Even attribute instances depending only on K-labeled arguments have to be labeled I, because inside the transformed area attribute instances don't have old values.

7. Experience

The following example will show a significant improvement in time and space requirement if demand attributes are used. The measures stated below were made for programs written in a toy language named BLAN.

BLAN is a Pascal-like, block-structured language with a statement part similar to BJ as described in [Wi79] . There is a symbol table† mechanism that supports static semantic and type checking. The tree transformer for BLAN generated by OPTRAN performs constant propagation and invariant code motion using data flow information collected by means of a special attributation. A description can be found in [Wi79] .

In this paper we will concentrate on a particular attribute; the so-called *constant pool*, represented by a list, where all variables with constant value are collected. Out of our OPTRAN specification, we will present only the transformations replacing a variable by a constant, if the variable occurring in a BLAN program has a value known to be constant. The latter property is checked by executing a pool membership test. The transformation has the following form:

transform <variable>
  if variable ∈ constant-pool
  into <constant>

The only need to know the pool's value will arise when evaluating the predicate. Because of this and the fact that the pool must be represented by a complex structure (e.g. a list of pairs *(name of variable, value of variable)*) we have chosen a complex attribute for it.

Figure 18 shows the benefit of choosing a complex attribute (version *C*) instead of a regular one (version *R*). Remember that attributes having complex type will only be evaluated when needed, whereas attributes of regular type will be reevaluated after each transformation. The examples were run under UNIX 4.2 BSD on a SUN 3/160 in single user mode. Note, that in a paged environment a saving in space will cause a runtime improvement in most cases.

Input:	Tree with 5399 operator nodes		
action	runtime in sec	space kB	version
initial evaluation	58.76 36.68	2704 1592	R C
after 43 transformations	74.48 57.72	2792 2112	R C

†using attributes of complex type

Input:	Tree with 20614 operator nodes		
action	runtime in sec	space kB	version
initial	234.08	10232	R
evaluation	154.34	5904	C
after 413	398.80	10792	R
transformations	237.68	7760	C

Figure 18

There is not only a substantial decrease in time, but also in space, since only attribute values at footholds are stored. Leaving out footholds would cause runtimes of several hours, as each entry in the pool depends on symbol table information and the symbol table itself is evaluated by demand, too.

8. Conclusion

The attribute evaluation and reevaluation scheme of the OPTRAN system was described. It combines data driven and demand driven (re)evaluation and allows efficient space management depending on user specification of footholds for attribute value storage. OPTRAN runs on VAX and SUN under UNIX 4.2 BSD. Comparative figures were given for nontrivial examples using purely data driven evaluation and demand driven evaluation with footholds.

9. Acknowledgements

Thanks go to Beatrix Weisgerber for her continuous support in the design and implementation of OPTRAN and many discussions about the subject of this paper. Heiner Tittelbach implemented the identity classes [Ti86] in OPTRAN.

Appendix A

G = {N, T, P, Z}
N = {AXIOM, DECLLIST, DECL, STATLIST, STAT, ID}
T = {var, begin, end, ;}
P = { AXIOM ::= var DECLLIST begin STATLIST end ,
DECLLIST ::= DECLLIST ; DECL ,
DECLLIST ::= empty ,
DECL ::= ID ,
STATLIST ::= STATLIST ; STAT ,
STATLIST ::= empty ,
STAT ::= ID}

Attr = Inh ∪ Syn, Inh = {is}, Syn = {ss, sd}

AXIOM ::= var DECLLIST begin STATLIST end
- $sd^{AXIOM} = sd^{STATLIST}$
- $is^{DECLLIST} = emptyset$
- $is^{STATLIST} = ss^{DECLLIST}$

DECLLIST ::= DECLLIST ; DECL
- $is^{DECLLIST_1} = is^{DECLLIST_0}$
- $ss^{DECLLIST_0} = union\ (ss^{DECLLIST_1}, ss^{DECL})$

DECLLIST ::= empty
- $ss^{DECLLIST} = is^{DECLLIST}$

DECL ::= ID
- $ss^{DECL} = setof\ (idno^{ID})$

STATLIST ::= STATLIST ; STAT
- $is^{STATLIST_1} = is^{STATLIST_0}$
- $is^{STAT} = is^{STATLIST_0}$
- $sd^{STATLIST_0} = and\ (sd^{STATLIST_1}, sd^{STAT})$

STATLIST ::= empty
- $sd^{STATLIST} = true$

STAT ::= ID
- $sd^{STAT} = memberof\ (idno^{ID}, is^{STAT})$

References

[Al81] Alblas, H., "A Characterization of Attribute Evaluation in Passes", *Acta Informatica*, (16) pp. 427 - 464 (1981).

[BG86] Bertling, H. and H. Ganzinger, "A Structure Editor Based on Term Rewriting", Conference Pre - Prints, Esprit Technical Week (23 - 25 Sept. 1985).

[Co84] Courcelle, B., "Attribute Grammars: Definitions, Analysis of Dependencies, Proof Methods", pp. 81 - 102 in *Methods and Tools for Compiler Construction*, ed. B. Lorho, Cambridge University Press

(1984).

[DJL86] Deransart, P., M. Jourdan, and B. Lorho, "A Survey on Attribute Grammars - Part 1: Main Results on Attribute Grammars", Rapports de Recherche, INRIA - Centre de Rocquencourt, Le Chesnay Cedex (January 1986).

[EF82] Engelfriet, J. and G. File, "Passes, Sweeps and Visits in Attribute Grammars", INF - 82 - 6, Twente University of Technology (1982).

[En84] Engelfriet, J., "Attribute Grammars: Attribute Evaluation Methods", pp. 113 in *Methods and Tools for Compiler Construction*, ed. B. Lorho, Cambridge University Press (1984).

[GPSW86] Greim, M., St. Pistorius, M. Solsbacher, and B. Weisgerber, "POPSY and OPTRAN - Manual", PROSPECTRA - Project, S.1.6 - R - 3.0, Universität des Saarlandes, Saarbrücken (1986).

[Jo84] Jourdan, M., "Les Grammaires Attribuees: Implantation, Application, Optimisations", Ph.D. Thesis, Universite Paris VII (1984).

[Ka80] Kastens, U., "Ordered Attribute Grammars", *Acta Informatica*, (13) pp. 229 - 256 (1980).

[KW76] Kennedy, K. and S.K. Warren, "Automatic Generation of Efficient Evaluators for Attribute Grammars", *Conf. Record of 3rd Symposium on Principles of Programming Languages*, pp. 32 - 49 (1976).

[Kn68] Knuth, D.E., "Semantics of context - free languages", *Math. Systems Theory* 2 pp. 127 - 145 (June 1968).

[Kr74] Kron, H.H., "Practical Subtree Transformational Grammars", Master Thesis, University of California, Santa Cruz (1974).

[Mö85] Möncke, U., "Generierung von Systemen zur Transformation attributierter Operatorbäume - Komponenten des Systems und Mechanismen der Generierung", Dissertation, Universität des Saarlandes, Saarbrücken (1985).

[Mö86] Möncke, U., "Grammar Flow Analysis", ESPRIT: PROSPECTRA Project Report S.1.3 - R.2.1 (1986).

[Ni83] Nielson, H.R., "Computation Sequences: A Way to Characterize Subclasses of Attribute Grammars", *Acta Informatica*, (19) pp. 255 - 268 (1983).

[Re82] Reps, Th., "Generating Language - Based Environments", Ph.D. Thesis, Cornell University (1982).

[Ti86] Tittelbach, H., "Effiziente Attributspeicherverwaltung für ein baumtransformierendes System", Diploma Thesis, Universität des Saarlandes, Saarbrücken (1986).

[We83] Weisgerber, B., "Die Baumanalyse und Untersuchungen zu Transformationsstrategien", Diploma Thesis, Universität des Saarlandes, Saarbrücken (1983).

[Wi79] Wilhelm, R., "Computation and Use of Data Flow Information in Optimizing Compilers", *Acta Informatica* 12Springer-Verlag, (1979).