Optimizing Implementation of Aggregates in the Compiler Specification System MAGIC [1]

A. Poetzsch-Heffter

Technische Universität München
Institut für Informatik
Arcisstraße 21
8 München 2
Federal Republic of Germany
poetzsch@lan.informatik.tu-muenchen.dbp.de

Abstract

The paper describes the implementation concept including optimizing transformations of the aggregate handling in the MAGIC System. The MAGIC System is a system for specification and rapid prototyping of compilers developped at the Technical University of Munich. One of its main features is a powerful functional specification language based on an extension of attribute coupled grammars [GaG84]. For the specification of structured symboltables, the language provides the generic abstract datatype *aggregate*.

As the user may handle aggregates like any other values, the system must provide the mapping of aggregate values to objects in the storage and of functions to storage changing procedures. This optimizing implementation mapping consists of three parts. First, all aggregate occurrences are determined and their dependencies are analysed. Then the algorithm tries to refine the attribute dependencies, so that reading operations to an aggregate-valued attribute preceed writing operations to this attribute. Finally, the functions are replaced by the corresponding operations that operate on a shared hashtable.

1. Introduction

The paper describes the implementation concept including optimizing transformations of the aggregate handling in the MAGIC System. The MAGIC System is a system for specification and rapid prototyping of compilers developped at the Technical University of Munich. Its main features (compared with other compiler generating systems) : It has a graphical user interface to support the specification process and to visualize compilations, a functional specification language based on an extension of attribute coupled grammars [GaG84], and an interpreter to allow high-level debugging. A short overview of the system (as far as needed for this work) is given in chapter 2; a more detailed description can be found in [KLP88].

[1] Support for this research was provided by the Siemens AG München ZTI SOF 2

1.1. The Problem

The major task in the development of a compiler front end is the design of a suitable symboltable; this especially holds for big languages like Ada or Chill where the front end is about half of the whole compiler (c.f.[Bjö80] p.IX) and a high-level specification with an efficient implementation of symboltables is particularly necessary. The existing compiler generating systems don't provide special means for the specification of symboltables:

- Either the user has to program imperative semantic actions getting efficient but very complex solutions which often even violate the paradigm of the specification formalism (cf.[BaT84])
- or the user specifies the symboltable by using functional list structures loosing efficiency (cf.[UDP82]).

The MAGIC Specification Language provides the generic abstract datatype *aggregate* with functions like enter, lookup, etc.. Aggregates are mainly used to specify structured symboltables (e.g. in a block structured language, the symboltable is a list of aggregates). As the user may handle aggregates like any other values (figure 1 shows a typical situation, where an aggregate value is used for the calculation of two attributes), the system must provide the mapping of aggregate values to objects in the storage and of functions to storage changing operations (implementation mapping).

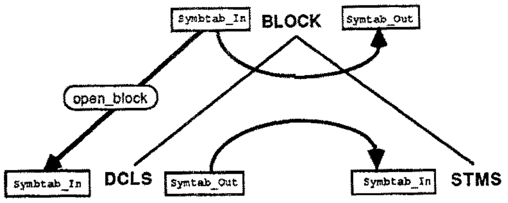


figure 1

The paper describes this implementation mapping and demonstrates how to use and refine attribute dependencies during generation time to get efficient implementations.

1.2. Informal Outline of the Approach

The user specifies the symboltable as a structure of aggregate types and uses the symboltable values as any other values in a purely functional manner. The system implements the aggregates by a shared datastructure with hashed lookup. In contrast to functional languages like ML [Har86], where the dynamic behaviour of a program is in general unknown, so that copy actions can't be avoided, functional attribute grammars allow effective optimization : The attribute flow can be calculated and even be influenced; and as the main application of aggregates is known, we can use heuristics where necessary.

The optimizing implementation mapping consists of three parts. First, all aggregate occurrences are determined and their dependencies are analysed. Then the algorithm tries to refine the attribute dependencies, so that reading operations to an aggregate-valued attribute preceed writing operations to this attribute. Finally, the functions are replaced by the corresponding operations that operate on a shared hashtable. Aggregate values are represented by access information to the hashtable. The algorithm for the implementation mapping is presented in chapter 3.

1.3. Related Work

The work is related to different papers in the area of functional languages and attribute grammars. Especially, four aspects in the treatment of attribute grammars have influenced our work, namely storage optimization [Gan79], nonlocal attribute handling [Räi86], lifetime analysis for attributes [Kas87], and incremental evaluation for aggregate values [HoT86].

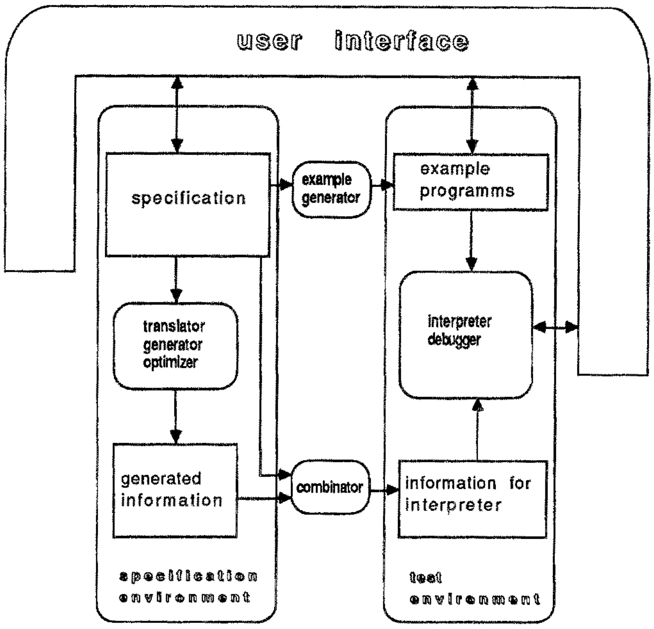
2. The MAGIC System and its Specification Language

2.1. The MAGIC System

The MAGIC System is a tool for the interactive specification and development of compilers. Compared with other compiler generating systems, it stresses the specification and testing process. For this purpose, the MAGIC System provides :

- a modular and powerful high-level specification language based on attribute grammars (introduced in section 2.2).
- a graphical and interaccive user interface to most components of the system supported by a consistency check mechanism.
- generator, interpreter and a graphical debugger enabling fast and high level testing of the specification as well as a rapid prototype implementation of the compiler.

The following figure gives a rough overview of the system:



The user interface supports the specification process by enabling users to directly edit graphical representations of attribute grammars and the testing phase by debbugging attribute evaluation on the same level. The main aspects of the graphical representation of attribute grammars will be introduced together with the specification language in the following section.

2.2. The MAGIC Specification Language

This section introduces those features of the MAGIC Specification Language (*MS*) needed for the following central chapter of this paper; a complete description can be found in [Poe88].

2.2.1. The Type Concept

MS has a uniform type concept to describe syntax trees and attribute values. Besides standard types and subtypes thereof, there are structured types (for the description of syntactical productions and variant records resp.), list types, and aggregate types. To illustrate the usage of types and their values, we stepwise introduce a miniature programming language minPL and specify a function *simple_sem_analyse*. This function computes a table that records the number of all occurences of an identifier where it was not visible. The following figure gives the abstract syntax of minPL :

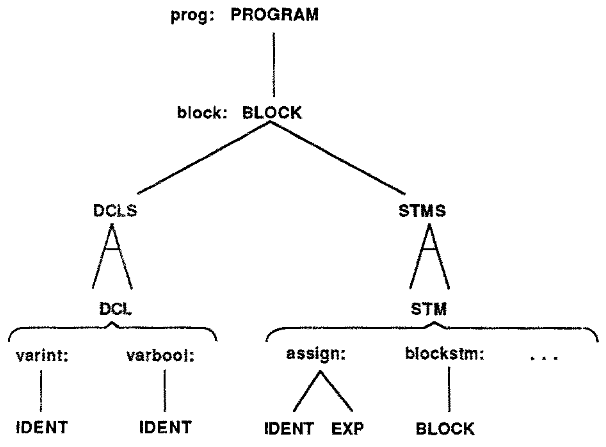


figure 2

As shown in the figure, a minPL program consists of a block that itself consists of a declaration list and a statement list. minPL provides integer and boolean variables. Every used identifier must have been declared as in known block structured languages. For brevity, we ommit other statements and the specification of expressions.

Figure 2 defines the grammar types PROGRAM, BLOCK, DCL, STM, and the list types DCLS and STMS, as well as the constructors (in figure 2 prog,varint,varbool,..) to construct the trees/terms of those types, to name the productions, and as discriminators of the defined type.

2.2.2. Lists and Aggregates

As shown in figure 2, the abstract syntax of a programming language can be defined by grammar types and list types. The manipulation of lists is done by functions like emptylist (denoted by <>), concatenation (.+.), and makelist (<.>).

For the specification of symboltable mechanisms and comparable tasks, the system provides aggregate types [BaW81]. E.g. to specify the function *simple_sem_analyse*, we need a symboltable that records the declarations. Figure 3 shows the definition of the type SYMTAB. Values of type SYMTAB are lists of aggregates of type LOCAL_SYMTAB. Each local symboltable is an aggregate with keytype IDENT and entrytype VARTYPE. VARTYPE is a type with the two constants vint and vbool to record the type of the found declaration.

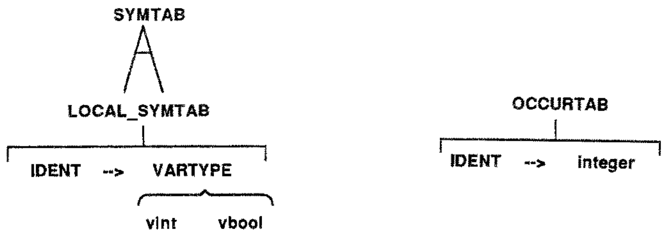


figure 3

Aggregates are constructed and handled by the following functions: empty_grex (denoted by []), enter (. &(.,.)), lookup (.[.]), isdefined, etc.. Their usage is illustrated in the following section.

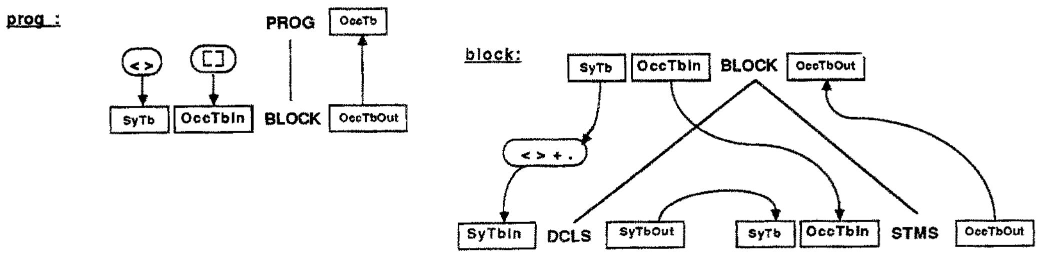
2.2.3. Functional Abstraction

In the MAGIC Specification Language, there are four kinds of functions:

- Standard functions: They are provided by the language to handle the standard and generic datatypes.
- External functions: They are an interface to the programming languages C and Pascal. They are not further discussed in this paper (c.f.[Poe88]).
- Applicative functions: They are recursively defined functions as *rec_isdefined* in the example below.
- Attributive functions: They are defined by means of an attribution as the function *simple_sem_analyse* in the example. They are an generalization of attribute coupled grammars [GaG84].

To continue the example and the illustration of aggregates, the following figure shows the specification of the function *simple_sem_analyse* consisting of the functionality and the corresponding attributions:

function simple_sem_analyse (PROG p) OCCURTAB:



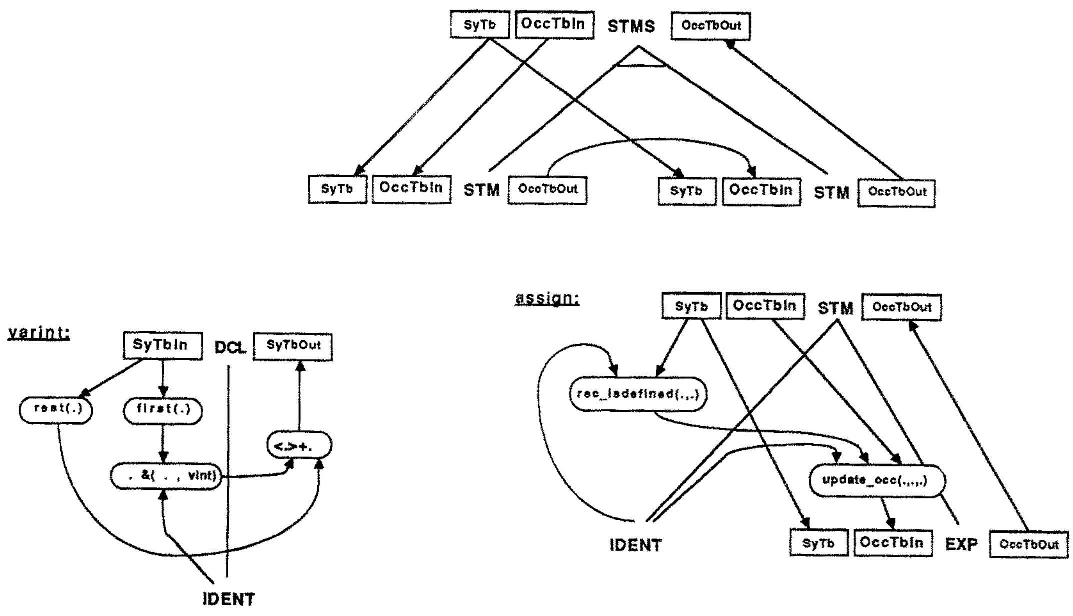


figure 4

For brevity, the trivial attribution schemes for DECLS and blockstm are left out; the attribution scheme for varbool is like that for varint.

By this kind of functional abstraction, attribute grammars become a flexible means for modular function specification. They can be used in all places where syntax-directed computation is needed: For the specification of a whole compiler phase as well as for the specification of semantic actions (not shown in the example).

Finally, we give the definitions of the predicate *rec_isdefined* that checks whether an identifier has been declared and the function *update_occ* that updates the occurrence table if the is_decl parameter is true:

```
function rec_isdefined  ( SYMTAB st; IDENT id ) boolean:
    if   is_<>(st)
      then    false
      else if  isdefined( first(st), id )
              then    true
              else    rec_isdefined( rest(st), id )
            end
    end
```

```
function update_occ ( boolean is_decl; OCCURTAB ot; IDENT id ) OCCURTAB:
    if  is_decl
      then   ot
      else if  isdefined( ot, id )
           then   ot &( id, ot[id]+1 )
           else   ot &( id, 1 )
         end
    end
```

3. Implementation of Aggregates

The following four sections describe the implementation of aggregates in the MAGIC-System. First, the target datastructure for aggregates is presented. Then, we explain the analysis of attribute equations and the algorithm for a slightly restricted language. Finally, we show the extensions for the unrestricted case.

3.1. Implementing Functions by Storage Changing Procedures

To implement the specification language of MAGIC, each function expression in an attribute equation has to be substituted by a storage changing procedure or function call and additional operations. Furthermore, datastructures must be given for the types of the language.

We implement aggregates by a shared hashtable. The following dependency graph gives an idea of what can be shared and motivates the chosen datastructure.

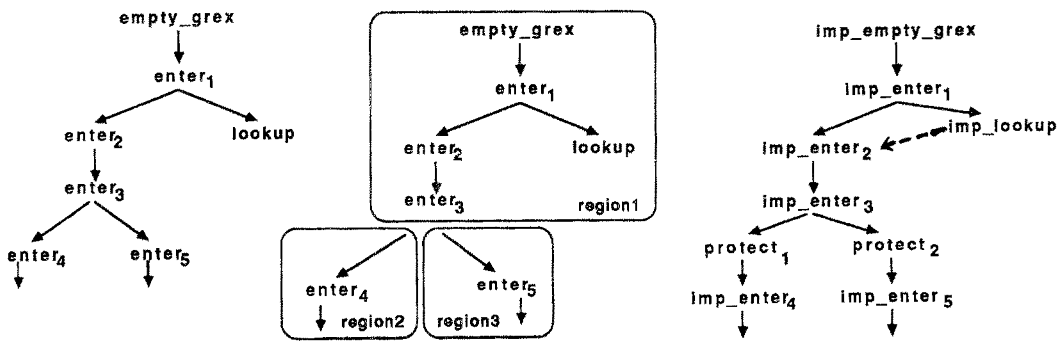


figure 5

As illustrated in figure 5, every aggregate value is computed by the function *enter* starting from an initial value (here *empty_grex*). If we can execute the *lookup*-operation before the *enter*$_2$-operation, then *enter*$_1$, *lookup*, *enter*$_2$ and *enter*$_3$ can operate on the same table, whereas the resulting table has to be protected from the entries of *enter*$_4$ and *enter*$_5$ in order to get correct tables after these entries; i.e. son nodes always share the table of their father. Figure 5 shows the corresponding partition of the dependency graph reflecting the sharing. E.g. the table after *enter*$_4$ consists of region 1 and region 2. The third part of figure 5 shows the dependency graph with the corresponding imperative procedures. Before we give their definitions, we describe the underlying datastructures and functions :

- A global hashtable HashTable and a global region counter RegCount
- A injective function *code_key*: KEY_DOMAIN x NAT —> HASH_KEY that codes the aggregate key and the region number into a key for the hashtable
- For each aggregate value a record AGGR with two components: One that records the corresponding list of region numbers (RegList); the other holds the boolean value IsProtected that records whether the aggregate value is protected (see below)

The functions are implemented by the following imperative procedures:

```
enter:
  procedure imp_enter ( AGGR a; KEY k; ENTRY e ) AGGR:
     if    a.IsProtected
        then    hash_tab_enter( code_key(k,RegCount) , e );
                var AGGR  avar := new(AGGR);
                avar.RegList     := append( RegCount, a.RegList );
                avar.IsProtected := false ;
                RegCount       := RegCount + 1 ;
                avar
        else    hash_tab_enter( code_key(k,~~RegCount~~ top(a.RegList)) , e );
                a
     end

empty_grex:
  procedure imp_empty_grex  () AGGR:
     var AGGR avar := new(AGGR);
     avar.RegList     := make_list( RegCount );
     avar.IsProtected := false ;
     RegCount       := RegCount + 1 ;
     avar
     end

lookup:
  procedure imp_lookup  ( AGGR a; KEY k ) ENTRY:
     imp_rec_lookup( a.RegList, k ) ;

  procedure imp_rec_lookup  ( REG_LIST regl; KEY k ) ENTRY:
     if  isempty( regl )
        then not_found
        else var e :  ENTRY_TYPE;
              e :=  hash_tab_lookup( code_key(k,first(regl)) );
              if  e == not_found
                 then     imp_rec_lookup( rest(regl), k )
                 else     e
              end
     end

The operation to protect hashtable states:
  procedure protect ( AGGR a ) AGGR:
     a.IsProtected :=  true ;
     a
```

3.2. Analysis of Attribute Equations

The aim of the approach is to replace the expressions on the right hand side of the attribute equations by the corresponding operations of the previous section and to refine the attribute and functional dependencies. As we want to perform static optimizations (i.e. at generation time), the actual dependency graph of the aggregate values is not available. But in contrast to functional programming languages like ML (cf.[Har86]), we posssess a very good approximization of it, namely the patterns it consists of: The dependency graphs of the grammar productions.

To concentrate on the main ideas, we introduce some simplifications; the general case will be discussed in section 3.4 :

- nested expressions in attribute equations are assumed to be broken off by using auxiliary attributes
- output attributes may not be used in the production
- structured types based on aggregate types are not allowed (e.g. a type like SYMTAB (see figure 2) is then forbidden)

With these restrictions, we get the following central definition :

Definition:

The *production dependency graph* (PDG) $\mathbf{G}_P$ = (V,E) of an attributed grammar production P consists of a set V of labelled vertices, a set E of labelled edges, and the functions

source,target : E —> V

to denote the incidence relation:

- V equals the set of attributes labelled by:
  attr_kind : V —> { in, out, aux } , telling the kind of the attribute;
  has_aggr_type : V —> boolean , telling whether the attribute will hold an aggregate value or not;
- E, *source*, and *target* express the attribute dependencies; E is labelled by:
  dependency_kind : E —> { i/o_attr_dep, aggr_passing_dep, aggr_using_dep, other_dep } ,
  in the following way:
  i) *dependency_kind*(e) = i/o_attr_dep
     if *source*(e) is an output attribute and
     target(e) is an input attribute of the same nonterminal.
  ii) *dependency_kind*(e) = aggr_passing_dep
     if *has_aggr_type*(*source*(e)) and
     has_aggr_type(*target*(e)) and
     the value of *source*(e) is passed to *target*(e)
     either without change or by an *enter* operation.
  iii) *dependency_kind*(e) = aggr_using_dep
     if the function that yields the value of *target*(e) has the aggregate valued attribute *source*(e) as a parameter and performs only *lookup* and/or *isdefined* operation on it.
  iv) *dependency_kind*(e) = other_dep in all other cases.

□

With the restrictions given above and because of strong typing in MS, the computation of V, *attr_kind*, and *has_aggr_type* is straightforward. The computation of E, *source*, *target* and *dependency_kind* proceeds as follows:

- compute the functional dependencies local to the production, and label the corresponding edges as described in the definition of PDG's. This is no problem for functions like *enter, lookup,* etc. or user defined functions that are not recursive. For recursive or attributive functions we compute a simple, but effective and of course correct static approximation of the dynamic dependencies:
  i) If the function has parameters and results of the same aggregate type, then label the corresponding edges by aggr_passing_dep.
  ii) If the function has parameters of aggregate type but the result is of another type, then label the corresponding edges by aggr_using_dep.
  iii) If the function has no parameters of aggregate type, then label the corresponding edges by other_dep.

  (cf. the discussion in chapter 4).
- compute the input/output-graph of the production as described in [KeW76] and label the resulting edges by i/o_attr_dep;

The production dependency graph is the abstract notation of the information that we need for the "value"-flow analysis in the following section.

3.3. The Algorithm

The algorithm has two parts. First, it refines the partial order given by the PDG in order to place aggregate using functions before aggregate passing functions with the same input aggregate (cf. *lookup* and *enter*$_2$ in section 3.1.). The second part of the algorithm labels those edges of the PDG where the procedure *protect* has to be inserted. Finally, the global frame of the algorithm will be discussed.

3.3.1. Refining the production dependency graph

The refinement of the PDG's is done to minimize the number of protect operations, i.e. to keep the region lists as short as possible. The refinement of one PDG proceeds as follows:

Let "<" be the partial irreflexive order implied by the PDG, i.e. v<v' if there is a sequence of edges that starts at v and ends at v';

1. Compute the set
   S^1_{refine} = { (v,e,U) ∈ V x E x (set of E) | has_aggr_type(v) and (e is the only edge with *source*(e) = v and *dependency_kind*(e) = aggr_passing_dep) and e' ∈ U <==> (*source*(e') = v and *dependency_kind*(e') = aggr_using_dep) }
2. Delete all tuples from S^1_{refine} where at least one element of the using set U depends on the value produced by the passing edge:
   S^2_{refine} = { (v,e,U) ∈ S^1_{refine} | there is no e' ∈ U: *target*(e) < *target*(e') }
   (In these cases the protect operation can't be avoided.)
3. Reduce the using sets in the tuples as follows:
   S^3_{refine} = { (v,e,U) ∈ V x E x (set of E) | there exists a (v',e',U') ∈ S^2_{refine} : v = v' and e = e' and U contains exactly the edges of U' that have maximal target vertices in *target*({e}∪U') w.r.t. "<" }
   (As we want to add dependencies between target vertices of the using edges and the target vertex of the passing edge e, we only have to consider "independent

edges").

4. Take *one* element (v,e,U) from S^3_{refine} and add other_dep edges between the vertices in *target*(U) and the vertex in *target*(e).
5. Delete the chosen element (v,e,U) from S^3_{refine} and proceed with step 2 until S^3_{refine} is empty.

□

Remarks:

i) The refinement process seems perhaps quite costly; but it is not, as even in big compiler front end specifications S^3_{refine} and the using sets rarely contain more than two elements.

ii) The choice of the element in step 4 is guided by heuristics which partially depend on issues concerning the extensions of the outlined algorithm (see section 3.4.). The main idea is to minimize the number of added dependencies. So we prefer elements with small using sets if a choice is necessary at all (cf. (i) above).

iii) Correctness of step 4: We have to show that adding dependencies doesn't destroy the partial order. This is proved by the following two lemmata:

Lemma 1
For each element (v,e,U) of S^3_{refine} holds that the targets of $\{e\} \cup U$ are not mutually comparable w.r.t "<", i.e.:
for all v, v' $\in$ *target*($\{e\} \cup U$) : $\neg\, v<v' \;\wedge\; \neg\, v'<v$.

Proof: Let v, v' be two targets of $\{e\} \cup U$ with $v < v'$;
Case v = *target*(e) : contradicts step 2.
Case v = *target*(U) : contradicts the reduction of the using set in step 3.

□

Lemma 2
Let PO(<) be a partial irreflexive ordering of V, $U \subset V$, and $p,u \in V \backslash U$ so that the elements of $U \cup \{p,u\}$ are mutually incomparable
(for all v,v' $\in U \cup \{p,u\}$: $\neg\, v<v' \;\wedge\; \neg\, v'<v$);
let PO'(<) be the transitive closure of $PO(<) \cup \{(u,p)\}$, i.e. the refinement of PO(<) by putting u before p.
Then PO'(<) is a partial irreflexive ordering of V and for all v,v' $\in U \cup \{p\}$:
$\neg\, v<v' \;\wedge\; \neg\, v'<v$;

Proof: i) PO'(<) is transitive by construction;
ii) PO'(<) is irreflexive: Suppose there is an element v with $v < v$; this implies $p < v$ and $v < u$; so we have $p < u$ in contradiction to the presupposition of this lemma.
iii) Suppose there are v,v' $\in U \cup \{p\}$ with $v < v'$; this implies $v < u$ and ($p = v'$ or $p < v'$) , which again contradicts the presupposition of this lemma.

□

Finite induction over U yields the desired result.

3.3.2. Insertion of the Protection Operation and Global Frame of the algorithm

After the refinement of a PDG, protection operations (see section 3.1.) have to be inserted where an aggregate value is passed to different attributes or where an aggregate value is possibly passed before it is used. In our formalism, we denote the insertion as the set of edges in question:

Let $\mathbf{G}_{refine}$ be the refined PDG; the set $E_{protect}(\mathbf{G}_{refine})$ is defined as follows:

$$\begin{aligned}
&E_{protect}(\mathbf{G}_{refine}) = \\
&\quad \{\, e \in E \mid \text{dependency\_kind}(e) = \text{aggr\_passing\_dep} \\
&\qquad \wedge (\exists\, e' \in E: \mathit{source}(e') = \mathit{source}(e) \\
&\qquad\quad \wedge (\ (\text{dependency\_kind}(e') = \text{aggr\_passing\_dep} \ \wedge \ e \neq e'\ ) \\
&\qquad\qquad \vee (\ \text{dependency\_kind}(e') = \text{aggr\_using\_dep} \\
&\qquad\qquad\quad \wedge \neg\ \mathit{target}(e') < \mathit{target}(e)\ )\ )\ \ )\,\}
\end{aligned}$$

Finally, the refined and augmented PDG is translated back to attribute equations whereby the functions of the functional specification are replaced by the corresponding imperative procedures, the protection operations are inserted, and control statements are added to respect the refinements.

Thus far, we have only discussed the treatment of single productions. It remains the problem, how to process the production list. As the refinement of one PDG causes a refinement of the i/o-graphs of the other productions, an efficient algorithm would be needed to incrementally recompute these i/o-graphs and to choose the next production to be refined.

Up to now, we can't give a really satisfactoring answer to this problem. We tackle it by starting with the root production, continuing with the productions of the sons, while recomputing the i/o-graphs as usual.

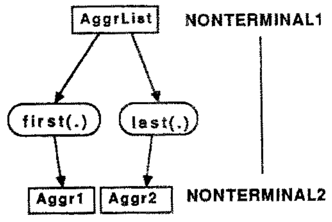
3.4. Raising the Restrictions

In section 3.2, we restricted the problem in order to concentrate on the main issues. The described algorithm is powerful enough to master aggregates like the occurence table in the example of chapter 2 : As it is never necessary to open a new region, the resulting occurence table implementation is as almost efficient as a hand coded implementation. But so far the algorithm can't deal with lists of aggregates like the symboltable of the example.

Whereas the first two restrictions are merely chosen to avoid technical overhead, the restriction to plain aggregates is a real simplification. In the general case, the PDG becomes quite more complex, as the attributes may contain several aggregates – in most cases a list of aggregates – so that the "value"-flow analysis has to keep track of sets of aggregates. To illustrate this, we shortly discuss two different cases:

i) In the *varint* production of the example, the incoming aggregate list is splitted in its first element and its rest; so the aggregate sets belonging to the *first* vertex and the *rest* vertex are disjoint and no protection operation has to be inserted.

ii) The following figure shows an attributed production where the first and last element of a list of aggregates are passed through. In this case, the aggregate values have to be protected, as they could be the same.



A more detailed and formal treatment of this problem can be found in [Ple88]. However, we expect further optimizations by using a more powerful "value"-flow analysis.

4. Conclusion

4.1. Summarizing Epilogue

We described the usage and implementation of aggregates in the compiler development system MAGIC and pointed out that having aggregates in the specification language has two seemingly contrary advantages:

- a convenient and powerful high-level specification facility
- efficient implementation

By closing this gap, we again realized the importance of a carefully designed typing mechanism for good language implementation.

Many details and improvements of the presented algorithm could not be discussed in this paper. Some of them should at least be mentioned:

- further optimization to avoid even more *protect* operations by taking advantage of the expression nesting
- treatment of *enter* operations in expressions that only "temporarily" change aggregates
- treatment of *delete* operations
- improvements to the analysis of recursive and attributive functions, especially if they have more then one aggregate as parameter
- storage management

4.2. Future Work

For the future, we envisage three other related topics:

- A better integration of this approach and other optimization techniques (and where possible a statistical support to guide the decisions).
- A careful analysis of the dependency between attribute evaluation strategy and aggregate optimization.
- A combination of this approach and globalizing transformations as described in [Räi86]. This would be particularly interesting to isolate certain classes of linearly behaving aggregate computations. For such classes, even more efficient implementations could be provided.

Acknowledgements

Thanks are due to A. Liebl and J. Knopp for helpful discussions, to Prof. J. Eickel for initiating and supporting the MAGIC Project and to F. Plenk; his diploma thesis was the basis for this work.

References

[BaT84] G. Bartmuß, S. Thürmel: MUG–Tutorial; Internal Report; Technical University of Munich; 84

[BaW81] F.L. Bauer, H. Wössner: Algorithmische Sprache und Programmentwicklung; Springer Verrlag; 81

[Bjö80] D. Björner: Towards a Formal Description of Ada; LNCS 98; Springer Verlag; 80

[Gan79] H. Ganzinger: On Storage Optimization for Automatically Generated Compilers; LNCS Vol. 67, pp 132–141

[GaG84] H. Ganzinger, R. Giegerich, et al.: Attribute Coupled Grammars; SIGPLAN 84 Symp. on Compiler Construction; 84

[Har86] R. Harper: Introduction to Standard ML; Dep. of Computer Science, University of Edingburgh; Technical Report ECS–LFCS–86–14

[HoT86] R. Hoover, T. Teitelbaum: Efficient Incremental Evaluation of Aggregate Values in Attribute Grammars; ACM Proc. of the 86 SIGPLAN Symposium on Compiler Construction

[Kas87] U. Kastens: Lifetime Analysis for Attributes; Acta Informatica 24; 1987, pp.633–651

[KeW76] K. Kenedy, S. Warren: Automatic Generation of Efficient Evaluators for Attribute Grammars; 3rd ACM Symposium on Principles of Programming Languages, Atlanta; 76

[KLP88] J. Knopp, A. Liebl, A. Poetzsch–Heffter: MAGIC – An Interactive Compiler Specification System; Internal Report; Technical University Munich; 88

[Ple88] F. Plenk: Optimierende Implementierungstransformation für funktionale Attributgrammatiken; Diplomarbeit; Technical University of Munich; 88

[Poe88] A. Poetzsch–Heffter: Report on the MAGIC Specification Language; to appear december 88

[Räi86] K.–J. Räihä: A Globalizing Transformation for Attribute Grammars; ACM Proc. of the 86 SIGPLAN Symposium on Compiler Construction

[UDP82] J. Uhl, et al.: An Attribute Grammar for the Semantic Analysis of Ada; LNCS 139; Springer Verlag; 82