A TRULY GENERATIVE SEMANTICS-DIRECTED COMPILER GENERATOR

Harald Ganzinger*, Robert Giegerich*
Institut für Informatik, Technische Universität
D-8000 München 2, Fed. Rep. of Germany

Ulrich Möncke+, Reinhard Wilhelm+
Fachbereich 10 Informatik, Universität des Saarlandes
D-6600 Saarbrücken, Fed. Rep. of Germany

1. Introduction

This paper describes semantic processing in the compiler generating system MUG2. MUG2 accepts high-level descriptions of the semantics of a programming language including full runtime semantics, data flow analysis, and optimizing transformations. This distinguishes MUG2 from systems such as YACC [Joh75], HLP [HLP78], PQCC [PQC79], or its own former version [GRW77] with respect to expressive power and convenience. In this respect, MUG2 comes close to semantics-directed systems such as [Mos76], [JoS80], [Set81], [Pau82]. In contrast to these, MUG2 is not a universal translator system where program independent semantic properties have to be evaluated at compilation time. The description concepts of MUG2 allow a far reaching separation of language vs. program dependent semantics, thus constituting a truly generative approach to semantics-directed compiler generation.

* Work partially supported by Sonderforschungsbereich 49 - Programmiertechnik

+ Work partially supported by DFG-project "Manipulation of Attributed Trees"

2. A Sketch of MUG2

In order to provide a little background, this section gives a rough and rather technical sketch of the MUG2 compiler generating system. Discussion of what makes MUG2 semantics-directed and still truly generative is spared out for later sections. Readers familiar with earlier versions of MUG2 from [GRW77] or [Gie79] may skip this section.

MUG2 is a system intended to provide automatic support for all phases in the implementation of programming languages. Particular emphasis was placed on mechanisms for an extensive amount of semantic processing, as is required by complex source language semantics and by optimization efforts. The basic design decision for MUG2 was not to try to supply general, standard solutions for specific purposes, such as a fixed intermediate language, symbol table mechanisms, or particular optimization algorithms, to which a particular language implementation would then have to be adjusted. Rather, MUG2 allows to specify these compilation subtasks and concepts on a rather high level, by providing an integrated ensemble of compiler description tools. Figure 1 gives a summary of the tools and concepts used in MUG2.

MUG2-generated compiler front-ends first construct an abstract parse tree, called program tree, as the intermediate representation of the program. Using abstract syntax rather than concrete syntax yields not only a significant cutdown in tree size, but also allows the semantic phases to operate on abstract syntax, as is customary e.g. in denotational language definitions.

Semantic analysis then proceeds by decorating the tree with semantic attributes, according to a description written in the attribute definition

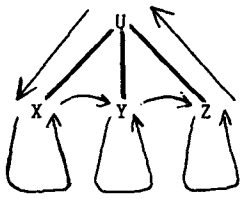
PHASE	lexical and syntactic analysis	construction of intermediate representation	semantic analysis (in several passes)	optimization	interpretation	(intermediate) code generation
MODULE	scanner and parser [LL(k) or LALR(k)], including error recovery	program tree contructor (constructs attribute-free abstract parse tree)	attribute evaluator (decorates abstract parse tree with semantic attributes)	transformations of the attributed program tree	attribute evaluator (evaluates functional attributes)	intermediate code generator (transforms the APT into a sequence of intermediate instructions)
TOOL	extended regular expressions and context-free grammar	string-to-tree grammar relating concrete and abstract syntax	attribute grammar (written in ADELE)	attributed transformational grammars (written in OPTRAN)	attribute definition language ADELE	code templates

Figure 1 : compiler/interpreter phases, modules, and description tools

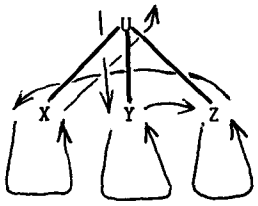
language ADELE. ADELE is embedded in PASCAL, allowing PASCAL data types as attribute domains, and PASCAL procedures for writing attribute definition rules.

ADELE restricts attribute dependencies to those which allow attribute evaluation in a number of passes over the program tree. "Pass", however, does not mean a strict left-to-right or right-to-left tree traversal, but more generally any depth-first tree traversal where the order of the visits to the subtrees of a given node is fixed and known at compiler generation time.

"Pass" in [JaW75]



"Pass" in MUG2 (e.g.)



Passes of this kind have been called sweeps in [EnF81]. ADELE attribute grammars are thus of type "n-sweep", $n \geq 1$, where the decomposition into the single sweeps has to be provided explicitly by the compiler describer.

Our experience is that this class of attribute grammars is sufficiently general [EnF81], [Joc81], yet retains the applicability of the efficient pass-oriented evaluation techniques of [JaW75], [Poz79], [Räi79], [JaP81]. Requiring an explicit separation into passes helps to keep compiler descriptions modular and comprehensible.

MUG2 allows the generation of compilers as well as interpreters. An interpreter definition may be written in ADELE directly. This would be the choice in language design research, or for language implementations where runtime efficiency is not a main objective.

Otherwise, more analysis needs to be done in preparation for intermediate or machine code generation. Data flow analysis, e.g. constant propagation or live variable analysis, can also be described in ADELE. For optimization, the program tree needs to be transformed. Series of tree transformations are specified in OPTRAN, a language based on attributed transformational grammars [GMW80]. There is a tight coupling between the attribute evaluation and transformation phases, discussed in section 5.

The MUG2 system currently running at the Technical University of Munich implements an older version of ADELE, already based on the principles to be discussed in the next sections, but still employing a different and somewhat clumsy notation for abstract syntax and attribute rules. This system has been tested on minor language examples, and has been used to generate a sophisticated formatter for mathematical formulas [JoW81]. We can say that the implementation has proven our

approach to be practical; its adaptation to the notational level as shown in this paper is under way. The transformation module is being implemented at Saarbrücken University.

So far, no generators for machine specific compiler parts have been incorporated in MUG2. However, a concept for the generation of machine specific ("peephole") optimizers from machine descriptions has been worked out [Gie82], and an implementation is under way.

3. The Attribute Definition Language ADELE

Scott/Strachey-style language definitions as being input to compiler generating systems described in [Mos76], [JoS80], [Set81], [Pau82] may be viewed as attribute grammars where complex functional semantic objects (e.g. state transformations) are associated with the program constructs as synthesized attributes (only), and are evaluated according to trivial bottom-up attribute dependencies: For each program construct it is specified how its semantics is determined from the semantics of its constituents. This structural induction technique is what denotational stands for. According to this view, execution of the program is a call to the state transformation function associated with the root of the program tree as a synthesized attribute. In which way the evaluation of this semantic function corresponds to a traversal of the program is not immediately apparent from the denotational definition. Thus, the structure of the information flow through a program is hidden from the compiler generators mentioned above. (It can be made explicit when deriving compiler definitions from language definitions by hand, but not all steps of this transition can be automated [Gan80].) This lack of information requires the generated interpreters to be based upon very general symbolic evaluation mechanisms that are known to be rather inefficient.

In contrast to this, realistic language interpreters consist of two parts. A compiler front-end performs syntactic analysis, semantic analysis, and the construction of an intermediate representation. This first step is independent of the input data for the program, corresponding to the static semantics of the program. The actual program execution is, then, simulated by performing the elementary operations of the program during traversing its intermediate representation according to the flow of control through the program.

Generating a reasonably efficient interpreter must include at least two kinds of optimizations:

1. Separation of static (i.e. compile-time) from dynamic (i.e. runtime) semantics at interpreter generation time should allow for generating a compiler front-end for the interpreter.
2. Making explicit the flow of control through the program should avoid the need for general symbolic evaluation mechanisms during interpretation.

Attribute grammars in the original sense of [Knu68] are a tool to meet this latter requirement. They use "finite" attributes to describe static (i.e. compile time) semantics. Finite means that attribute values can be explicitly calculated while traversing a graph which represents their dependencies. Under practical restrictions this graph needs not be constructed at compile-time. Rather, the program tree is traversed according to a strategy that can be completely determined at compiler generation time. However, using both inherited and synthesized attributes and almost arbitrarily complex attribute dependencies cannot make up for the restriction to nonfunctional attributes. It becomes impossible to describe the dynamic (i.e. runtime) semantics of programs, or its static approximations relevant for data flow analysis.

ADELE is an attribute grammar meta-language which, by providing a specific concept of functional attributes, combines the positive aspects of expressiveness and efficiency of both approaches. It is a tool for describing semantic analysis, data flow analysis, and dynamic semantics for a programming language, in a style that has been termed semantics-directed, and yet allows to generate efficient compilers and interpreters.

3.1. Overall Structure of ADELE Attribute Grammars

Attribute grammars in ADELE are structured into subgrammars called passes, where each pass is required to satisfy the 1-sweep property [EnF81], cf. above. The following example gives the structure of the ADELE attribute grammar for a PASCAL-like language.

```
Example 1.
begin
syntax
  ... abstract syntax of the source language ...
pass declarationHandling;
  ...  cf. examples 3,4,6
     componentInfo(...):(...);
     pass
        ... cf. examples 5,7
     end pass
end pass;
pass typeChecking;
  ... cf. examples 2,8
end pass;
iterative pass constantPropagation;
  ...
end pass;
unit constantFolding;
    ... cf. example 12
end unit;
iterative pass deadVariables;
  ... cf. example 10
end pass;
unit deadStatementElimination;
    ... cf. example 11
end unit;
pass interpretation;
     stateTrans(...):(...);
     pass
       ... cf. example 9
     end pass
end pass
end
```

An ADELE program consist of definitions of

- the abstract syntax of the source language,
- the sequence of attribute sub-grammars (in the above example for declaration handling, type checking, constant propagation, dead variables, interpretation) and intermediate transformation passes representing the sequence of compiler and interpreter phases.

The outer passes may contain inner sub-grammars (componentInfo, stateTrans), representing families of functional semantic objects, cf. 3.3. Attribute evaluation passes may be interleaved with program transformation passes (constant folding, dead variable elimination) specified as OPTRAN transformation units. Attributes being defined by a pass may be input to later passes and transformation units.

3.2. Basic Notation

The following example illustrates the ADELE-notation for attribute rules.

```
Example 2.
where  exp[0] -> exp[1] add exp[2]  do
  expType of exp[0] =
     begin if %expType of exp[1]% = intType
              and %expType of exp[2]% = intType
              then %expType of exp[0]%:= intType
              else ...
           ...
     end .
```

The example shows an attribute rule for deriving the type of an expression in the type checking pass. The **where**-clause gives the production of the abstract syntax to which this rule refers. Here, multiple occurrences of the same nonterminal symbol (exp) are distinguished by indexes (e.g. exp[1]). A synthesized attribute expType is assumed to have been declared in the declaration part of the type checking pass. Its domain is the (PASCAL-) enumeration type typeKinds, cf. example 4, including the constant intType. By saying expType **of** exp, expType is implicitly declared to be associated with the nonterminal exp. The lefthand side (left of "=") of the attribute rule gives the attribute occurrence whose evaluation is defined by the rule. The righthand side is, syntactically, a PASCAL procedure body, followed by "." as end marker. Attributes within the PASCAL text have to be enclosed by %-signs, otherwise they may be looked upon as ordinary PASCAL variables. They can be viewed as the (**var**-) parameters for the procedure body.

3.3. Functional Attributes

Attributes in ADELE may also take functions as their values, or as components of their value, where the value itself is a record. As this falls outside the PASCAL framework, ADELE provides mechanisms for the declaration of functional attribute domains, for defining, and for calling functional attributes. We will introduce each of these in turn by considering as an example the handling of complex declaration information.

In denotational language definitions, complex declaration information is represented by functions. E.g. record type denotations would be viewed as functions

componentInfo ≤recTypes, where
recTypes = [identifiers -> types],

mapping a field identifier of the record into the type of the corresponding field.

3.3.1. Definition of a Functional Domain

```
Example 3.
functype recTypes = (identifiers) : types
```

In order to incorporate functions as components of more complex PASCAL data structures, a functional domain can be used in a PASCAL type definition.

Example 4.
```
type typeKinds = (intType,recType,errorType);
     types = record case kind : typeKinds of
                    recType : (denotation : %recTypes%)
             end;
```

Technically, %recTypes% denotes the PASCAL representation of recTypes. Nothing is known about it to the user except that it is possible to assign variables of this type. Thus, the only meaningful manipulation of functional data on the PASCAL-level of an ADELE-program is passing it on. Note that types is defined recursively, since types occurs as result type of recTypes.

3.3.2. Defining and Creating a Function

The clue of having functional attributes is the way in which these functions are defined. Traditionally two views of what an attribute grammar specifies are in use. The first considers an attribute grammar to specify the decoration of parse trees with the values of the attributes. In this case, the attribute grammar defines a translation of parse trees into decorated parse trees. Alternatively, one might consider the (functional) attribute dependencies to be the objects of interest. In particular the dependencies between inherited and synthesized attributes of a nonterminal are important when considering the correspondence between attribute grammars and classical denotational definitions [ChM77], [Gan80].

This latter view is as follows. Given an attribute (sub-) grammar AG and a node X in the parse tree, AG **of** X (resp. the evaluator generated for AG) is a function mapping any tupel I of arguments to a tupel S of results as follows:

1) Initialize the inherited attributes at X with I.
2) Evaluate the AG-attributes in the subtree at X.
3) Deliver the (final) values S of the synthesized attributes at X as result.

After delivering the result, the attributes that have been evaluated in course of this call will vanish. Since the values of the inherited attributes at a node define the value of each attribute in the node's subtree in an unambiguous way, AG **of** X is in fact a function.
The next example defines the above mentioned functions for fields-subtrees of record type denotations. It will be

```
componentInfo of fields (i) =
   if << i names a fields component of type t >>
     then t else <<error>>.
```

Example 5.
```
componentInfo ( fieldId : identifiers )
              : ( fieldType : types );
pass
start fields; {the start symbol of the grammar}
where fields -> field do
fieldType of fields =
  begin if %fieldId of fields% = %id of field%
        then %fieldType of fields%:= %sType of field%
        else %fieldType of fields%.kind:= errorType
  end .

where fields[0] -> fields[1] ; field do
   ... { see next example } ... ;
end pass
```

The name of each field of a record is given by (the previously computed) id **of** field. If it is equal to the argument fieldId **of** fields of componentInfo, the (previously calculated) type information sType **of** field is the result of componentInfo in this case. componentInfo yields errorType otherwise.

The above defines, for each fields-node in the tree, a function componentInfo **of** fields, which may be called to retrieve information from this node's subtree - and may be passed on as an attribute value itself. The latter is shown in the next example.

Example 6.
```
where recordType -> record fields end do
   sType of recordType =
   begin %sType of recordType%.kind:= recType;
         %sType of recordType%.denotation
                        := %componentInfo of fields%
   end .
```

sType may now made to be the symbol table entry for type identifiers.

3.3.3. Evaluation of Functions

The operator **eval** initiates the evaluation of a function with given arguments.

eval$(f,i_1,\ldots,i_n,r_1,\ldots,r_m)$

denotes the assignment

$(r_1,\ldots,r_m) := f(i_1,\ldots,i_n)$

of the value of f at $i_1,\ldots,i_n$ to the result variables $r_1,\ldots,r_m$.

Example 7.
The other rule of the "body" of the grammar componentInfo contains a call to itself:

```
where  fields[0] -> fields[1] ; field  do
  fieldType of fields[0] =
   begin
   if %fieldId of fields[0]% = %id of field%
   then %fieldType of fields[0]%:= %sType of field%
   else {The  fieldId  may name a field of the initial
         fields-part}
        eval (%componentInfo of fields[1]%,
              %fieldId of fields[0]%, {input parameter}
              %fieldType of fields[0]% {result parameter})
   end .
```

The example shows that functions can be defined recursively.

The function componentInfo may be passed on (via symbol table attributes) as an attribute value to each occurrence of a variable of the record type. This provides complete information about the type where it is needed. The following example gives the semantic rule that uses the componentInfo- functions to perform type checking for selected components. An attribute varType (with domain types) is calculated accordingly:

Example 8.

```
where  selection -> var . fieldId  do
varType of selection =
begin
if %varType of var%.kind = recType
   then eval (%varType of var%.denotation,
              %id of fieldId%,
              %varType of selection)
   else %varType of selection%.kind:= errorType;
if %varType of selection%.kind = errorType
   then  error('selector not applicable')
end.
```

3.4. Description of Interpreters

The concept of functional attributes which ADELE provides is powerful enough to allow descriptions of interpreters in a natural way, modelling the runtime behaviour of the source programs. Thus, language specifications in MUG2 define the semantics of a language completely (though in an implementation-oriented way) with respect to both compile-time and runtime.

Example 9.
In the absence of gotos, a state transformation function describes the semantics of statements.

```
stateTrans ( initial : states ) : ( final : states );
pass
start ..., stats, ...;
    { In particular for any statement it is to be
      specified how the final state depends on the
      initial state. }
...
where  while -> while cond do stats  do
  initial of cond = initial of while;
  final   of while =
      var  s : states;
      begin  if %bvalue of cond%
             then
             begin
                  eval (%stateTrans of stats%,
                        %initial of while%, s);
                  eval (%stateTrans of while%,
                        s, %final of while%)
             end
             else %final of while%:=%initial of while%
      end .
...
end pass
```

If bvalue of cond evaluates to false, the stats-subtree need not be traversed, since none of its synthesized attributes is then referenced. On the other hand, the use of bvalue of cond makes necessary a traversal of the cond- subtree in each iteration through the while-loop.

Since it is possible to store functional objects such as state transformations into symbol table attributes, languages with procedures and/or gotos do not cause difficulties.

3.5. Implementation of Functional Attributes

Functional attribute values that occur during attribute evaluation always have the form A of X, where A is an attribute (sub-) grammar and X is a node in the parse tree. This is a consequence of the fact that no other function creating operations are provided. Thus any such value can be stored using two integers identifying the grammar A and the node X, respectively. Space consumption for functional attribute values is, therefore, very low.[1] For any attribute subgrammar such as A, a separate, efficient attribute evaluator is generated. Technically, the result is a PASCAL-procedure Psb(A) that has one additional input parameter which supplies the root of the subtree for which attribute evaluation is to be evoked upon an **eval**. (Psb(A) traverses this subtree depth-first, skipping those of the inner subtrees whose synthesized attributes are not referenced, and do, therefore, not contribute to the final result of the call.) Thus, the procedure call

---

[1] Coding semantic objects such as componentInfo into nonfunctional data would require space linear in the size of the record definition for any occurrence of the attribute.

$P_A(X,i_1,\ldots,i_n,r_1,\ldots,r_m)$
implements the **eval**-operation
eval (A **of** X, $i_1,\ldots,i_n,r_1,\ldots,r_m$).
This demonstrates that time and space consuming techniques such as symbolic evaluation of expression trees are not required to implement the functional data in ADELE.

We realize that our requirements concerning the generation of interpreters are satisfied:

1. The interpreter describer provides the separation between compile-time and interpretation time by structuring the attribute grammar into sub-grammars.
2. Describing state transformations as attribute dependencies makes efficient attribute evaluation algorithms available at interpretation time. The attribute flow represents the flow of control through a program. E.g. the fact that lists of statements are executed in textual order would be recognized from the left-to-right dependencies between the program state attributes initial and final.

It should be noted that the interpreters in order to not be too unrealistic require attribute space optimizations as proposed by [Räi79] or the use of pointers to represent the values of the state attributes initial and final. The latter solution would leave the responsibility for safe pointer usage to the language describer.

3.6. Definition of iterative compilation algorithms

Strong demands for the efficiency of target programs require optimizing compilers. In MUG2, optimizations are described by a kind of transformational grammars described below. A prerequisite is the availability of optimization information obtained by global data flow analysis. In contrast to the PQCC-project [PQC79] where a complete parameterized sequence of flow analysis, optimization, and code generation phases has been designed according to a series of pragmatic decisions, for MUG2 the availability of general description tools for these phases was the major design criterion. Consequently, the attribute grammar concept must allow the explicit description of both high-level and low-level global data flow analysis. Whereas the high-level problem can nicely be described by attribute grammars in the classic sense [Ros77], [Wil79], the low-level case implies the occurrence of circular attribute dependencies [BaJ78]. The latter requires to iteratively recompute attribute values.

In ADELE, attribute evaluation passes can be specified to employ certain kinds of circular attribute dependencies.

Example 10.
We describe the crucial part of dead variable analysis. It is assumed that use **of** cond (with domain **set of** identifiers) is the set of variables that occur in a while-condition or an expression.

```
iterative pass deadVariables;
  { The keyword iterative indicates that circular
    attribute dependencies are to be specified in this
    subgrammar. }

where assignment -> var := exp do
  sDeadVars of assignment =
     %iDeadVars of assignment%+%id of var%-%use of exp%;

where while -> while cond do stats do
  sDeadVars of while =
      %sDeadVars of stats% - %use of cond%;
  iDeadVars of stats =
      %iDeadVars of while% *
       %last sDeadVars of stats initially [1..maxIdNo]%
         - %use of cond%
...
end pass
```

The circular dependency of iDeadVars **of** stats on sDeadVars **of** stats is indicated by the prefix **last**. The **initially**-clause specifies that the full set of identifiers is the value for sDeadVars **of** stats the iteration process starts with.

3.7. Characteristics of Iterative Passes in ADELE

1) Erasing dependencies prefixed by **last** must yield a (noncircular) attribute subgrammar of 1-sweep type. This implies that if an iterative pass defines any circularities, at least one loop connecting edge of any cycle in the dependency graph is a **last**-dependency.
2) Attribute evaluation is iteratively repeated. If in the i-th iteration step an attribute **last** a **of** X is referred to, the value of a **of** X after the (i-1)-th step is taken. For i=1, the **initially**-clause applies.
3) Iteration stops, if for all of the **last**-attributes, the new values are equal to those obtained in the previous step.

Pass-oriented evaluation strategies traverse the parse tree depth-first where the subtrees of a node are visited in a order that depends on the attribute dependencies at that node. It is important to note that, because of 1), this represents a depth-first traversal of a spanning tree of the dependency graph. This order of

evaluating data flow attributes has been called rPostorder in [HeU75] and [AhU77]. Therefore, our attribute evaluation strategy is optimal among all iterative algorithms [Tar76] and "usually linear" [KaU76]. For non-structured control constructs, however, the attribute dependencies are merely an approximation of the flow graph.

4. The Optimization Phase

Optimization, like constant folding, invariant code motion, etc., modifies the internal representation of the program. With MUG2, such modification is described by an OPTRAN-program [GMW80]. When a full interpretative language definition is given, the correctness of the optimizations relative to this definition may be shown. In some cases, this proof can be carried out directly on the notational level of the compiler description.

4.1. Describing optimizations

Different optimizations are formulated as transformation units (T-units). They can be activated sequentially as passes over the program tree, or in a procedure-like fashion for subtrees of the program. The statements in a T-unit are transformation rules for attributed trees. Execution of the T-unit means application of its rules to the given program tree in a bottom-up strategy: Transformation starts at the leaves of the tree. Each node is inspected as to whether a rule is applicable at this node. One of the applicable rules is selected and the transformation is performed accordingly. Then, transformation resumes at a previously unvisited brother or else at the father of the current node. In this way, termination of the transformation process is guaranteed.

A rule is applicable if
- its input template matches some subtree of the program tree (syntactic match), and
- the enabling condition associated with the input template is satisfied by the attribute values in the matched subtree (semantic match).

Tree templates are parse trees for sentential forms of the abstract grammar. The roots of left and right side of a transformation rule must have the same terminal or nonterminal label. If a new terminal node is introduced in the righthand template of a rule, its lexical attribute must be set explicitly. Prefix notation is used for tree templates.

Example 11.
```
unit deadStatementElimination ;
  safe declarations, typeChecking, deadVariables;

 transform <stat, <assignment, var, exp>>
      into <stat, empty>
       if %id of var% in %iDeadVars of assignment%;

 transform <stat,<ifstat,exp,<stat,empty>,<stat,empty>>>
      into <stat,empty>;
   { semantically always applicable. }
 transform <stat,<ifstat,exp,stat,<stat,empty>>>
      into <stat,<ifstat,exp,stat>>
   ...
   ...
end unit
```

The above example assumes attributes id and iDeadVars to be evaluated in pass deadVariables. Furthermore it is specified (**safe**-clause) that the elimination of dead variables does not lead to inconsistencies (of a kind to be explained below) of attribute values that have been calculated in the declarations, typeChecking, and deadVariables passes.

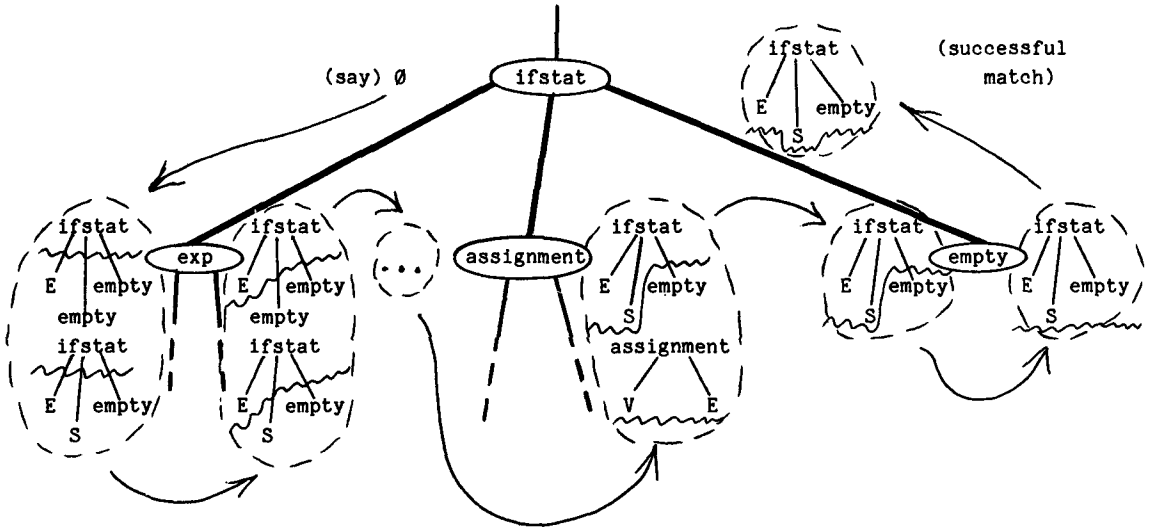
Example 12.
```
unit constantFolding
  safe declarations, typeChecking, constantPropagation;

   transform <exp,var> into <exp,const>
      where lexinf of const=
                value(%const_pool of var%,%id of var%)
      if %id of var% in %const_pool of var%;
   ...
   ...
end unit
```

It is assumed that the attribute evaluation pass for constant propagation has left the attribute const_pool at appropriate nodes. The rule introduces a new terminal node const, for which the lexical attribute **lexinf** is set to represent the constant value of the variable.

4.2. Tree Analysis and Transformation

The recognition of input template matches is the task of the tree analyser generated from an OPTRAN T-unit. In MUG2, tree analysis is seen in the framework of attribute grammars. We call the attributes in an attribute (sub-) grammar for tree analysis template recognition attributes. They represent sets of templates and subtemplates that have been recognized as (partially) matching the



The trivial syntactic rules for stat have been omitted.
The matched sub-templates are those above ~~~ .

Figure 2 : Attribute Evaluation for Tree Matching

subtree under analysis. Different tree analysis strategies correspond to different attribute evaluation strategies. Bottom-up tree analysis [Kro75] (which MUG2 uses) leads to an S-attributed grammar, using only synthesized attributes. An alternative algorithm, called LR-tree analysis has been developed. It starts at the root and performs a top-down depth-first tree walk, calculating the matching information according to an L-attributed [LRS73] grammar.

One snapshot of LR-tree analysis for the T-unit dead statement elimination is given in figure 2. LR-tree analysis provides more information about the context of a subtree than bottom-up analysis. This allows an early recognition of those sub-templates that cannot match at an inner node of the subtree. As a consequence, the size of the template matching attributes decreases considerably.

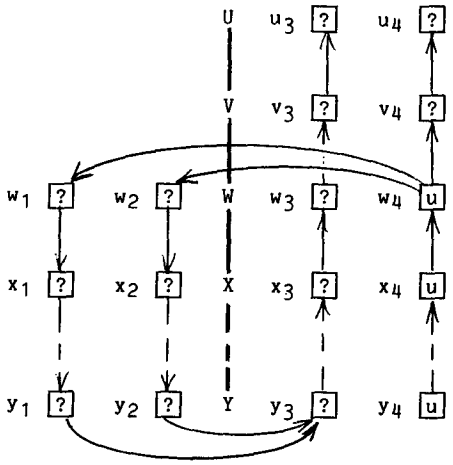
Obviously, the attribute dependencies needed for tree analysis are well within the class which ADELE permits.

Neither the representation for the attributes for the tree matching, nor the functions manipulating them need to be known to the compiler designer. They are automatically generated for each T-unit.

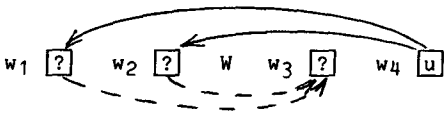
5. Co-operation of Attribute Evaluation and Transformations

In general, the transformation of an attributed tree may lead to a tree with attribute inconsistencies [DRT81], i.e. attributes of neighbouring nodes whose values do no longer satisfy the semantic rules of the original grammar. In a syntax-directed editor, for example, arbitrary subtree replacements are possible, hence incremental attribute evaluation is used extensively to correct attributes in the modified program tree. An optimizing compiler is essentially different: it only performs semantic equivalence transformations of the program. Where attributes encode some semantic property of the program that is not disturbed by a transformation, their values will not be invalidated either, and no re-evaluation is required. A typical example for this are attributes containing data flow information: constant folding does not make any variables non-constant that were constant before folding. Contrary, template-matching attributes, which are exclusively concerned with the syntactic shape of the tree are very sensitive to tree transformations, and need to be updated after each transformation.

In the subsequent sections, we first sketch an algorithm for incremental updating of attributes. Then we study the situation where no re-evaluation is needed in more detail. Finally, we indicate the



global situation



view at node W

---> : projections of global dependencies

possible choices for updating sequences:

i) $w_1\ x_1 \ldots y_1$ (!) $w_2\ x_2 \ldots y_2\ y_3 \ldots x_3\ w_3 \ldots$

ii) $v_4\ u_4 \ldots$ (!) $w_1 \ldots$

iii) $w_1\ w_2\ x_1\ x_2 \ldots y_1\ y_2\ y_3 \ldots x_3\ w_3\ v_3\ v_4\ u_3\ u_4$

Figure 3 : Snapshot of incremental attribute evaluation

pragmatic decisions to which these considerations have led for MUG2.

5.1. Attribute Re-evaluation

Attribute re-evaluation, which may be called upon over and over again during program transformations, must be designed very carefully in order to avoid unbearable overhead. Given that our basic data structure is the program tree, and that attributes are accessed via the nodes of the tree, a realistic criterion for an optimal incremental attribute evaluation algorithm must include the following:

a) An attribute should be re-evaluated at most once, and only when some attribute it depends on has changed its value. (Reps' algorithm [Rep82] satisfies this criterion.)
b) The number of visits to different tree nodes (i.e. the cost of traversing the tree in order to access attributes) should be minimized.
c) The (time and space) cost for controlling the re-evaluation process (e.g. manipulations of attribute dependency graphs made in order to meet a) and b)) must also be taken into account.

With respect to c), one difference of our algorithm to that of Reps is rather than modelling the affected section of the global attribute dependency graph explicitly, it labels attribute instances in the tree with "p" for "found to be preserved", "?" for "not regarded yet", or "u" for "updated".

With respect to b), take a look at figure 3. There, choices i) and ii) violate criterion b) as indicated by (!).

In [Mön82] an incremental attribute evaluation algorithm using the p-?-u labelling is developed, which tries to minimize overall cost by scheduling attribute re-evaluation according to two principles:

a) Select, for updating, attributes of the current tree node as long as possible.
b) When the current node has to be left, select a visit to the father or a son node which will yield additional "u"- or "p"-attribute instances at the current node.

In making the local decision in b) for the current node, the algorithm uses a "view" of the global dependencies between the current node's attribute instances. These views are a parameter to the algorithm - they may be "characteristic graphs" [CoH79] or approximations of those, "IO-graphs" of

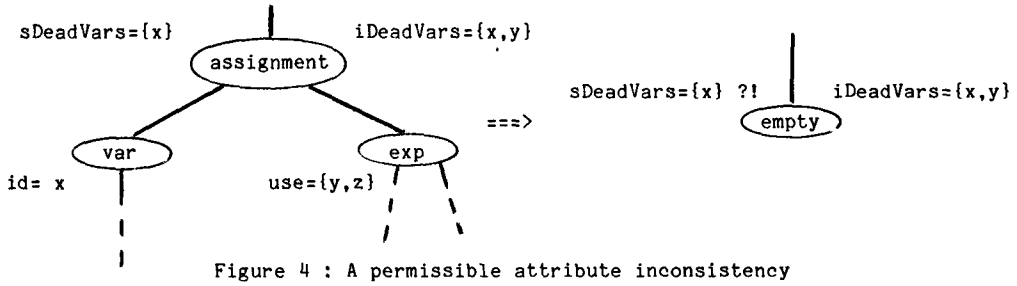


Figure 4 : A permissible attribute inconsistency

[KeW76], or the superior/subordinate graphs of [Rep82]. Depending on the class of attribute grammars given, part of the graph constructing and manipulating effort can be removed from attribute evaluation to compiler generation time.

5.2. Relaxing the Attribute Consistency Requirement

Figure 4 gives a simple example of an attribute inconsistency resulting from an application of the first rule of the T-unit dead assignment elimination to a program.

Clearly, sDeadVars **of** empty should be {x,y} and is therefore inconsistent in the sense of [DRT81]. But note that the value {x} taken over from the old sDeadVars **of** assignment is semantically not incorrect at this point - it is merely weaker than what we would obtain by re-evaluation. In the standard lattice-theoretic setting of data flow analysis, we would call the inconsistent value an approximation [CoC79] of the consistent one.

We say that some semantic information, collected in some attributes, is invariant wrt. a transformation rule R, if after any legal application of R, the old attribute values are still consistent in the transformed tree.

We say that it is safe wrt. R, if the old values may become inconsistent, but are an approximation of some consistent set of attribute values. (These notions are developed formally in [GMW81].)

For non-flow attributes (e.g. type information), safeness and invariance falls in one, as the only meaningful approximation is identity.

In order to prove that deadVars-information is safe wrt. the rule applied in Fig. 4, we would have to show that

id **of** var **in** iDeadVars **of** assignment =>
sDeadVars **of** assignment **in** sDeadVars **of** empty
where iDeadVars **of** empty =
iDeadVars **of** assignment,

i.e. the enabling condition of this rule implies that the old attribute values approximate those that would be obtained by re-evaluation.

This is easily confirmed from the attribute evaluation rules for assignment and empty.

Safeness guarantees correct handling of inconsistently attributed trees over a series of transformations. So the compiler may be more efficient, because re-evaluation is not mandatory after each step. If performed, however, it might disclose further opportunities for optimization. Where safeness is violated, the re-evaluation mechanism must be called upon.
Invariance is an even more preferable property. It guarantees preservation of consistency a priori, and as a consequence, transformation phases based on invariant information are exhaustive.

5.3. Coordinating Attribute Evaluation and Transformations in MUG2

The above considerations have led to a series of pragmatic decisions that were taken for MUG2:

1) During the application of a T-unit, template matching attributes are updated incrementally.
2) New tree nodes introduced by the output template of a transformation are attributed according to the rules of the attribute grammar.
3) Attributes referenced by the current transformation pass must be safe with respect to the rules of this T-unit, and must be declared to be so in the **safe**-clause of the T-unit.
4) Attributes to be referenced in later passes may also be declared to be safe. Otherwise, they will be re-evaluated non-incrementally before their next usage.

Upon the user of MUG2, we impose the responsibility to prove that the attributes listed to be **safe** are in fact safe for the given T-unit. (This could be enforced at compilation time, if the safeness criterion is identity, or can be inferred from explicitly given termination criteria for iterative passes. At the moment, we are not planning to include this compiler testing option.)
By these rules, the writer of a compiler description is advised that attributes that are sensitive to optimizing transformations, but are not needed for them, should not be evaluated until after the transformations (which appears reasonable anyway). He is required to devise separate T-units for transformation rules which could mutually destroy the information they rely on.

For our example, constant propagation attributes are invariant with respect to constant folding, while deadVars-attributes are only safe. Constant propagation attributes are not safe wrt. elimination of redundant assignments. This implies that both T-units do not require intervening re-evaluation, and that they cannot be combined in a single T-unit. Also, it would be possible to do deadVariables prior to constantFolding (including deadVariables in the **safe**-clause of constantFolding). Then, however, deadStatementElimination may miss some chances introduced by constantFolding. So, the arrangement shown in example 1 is the stronger one.

6. REFERENCES

[AhU77] Aho, A.V., Ullman, J.D.: Principles of compiler design. Addison-Wesley, 1977.

[BaJ78] Babich, W.A., Jazayeri, M.: The method of attributes for data flow analyis, part I: Exhaustive analysis. Acta informatica 10 (1978) 245-264.

[Boc76] Bochmann, G.V.: Semantic evaluation from left to right. CACM 19 (1976).

[CoC79] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. POPL 6, 1979, 269-282.

[CoH79] Cohen, R., Harry, E.: Automatic generation of near-optimal linear-time translators for non-circular attribute grammars. POPL 6, 1979, 121-134.

[ChM77] Chirica, L.M., Martin, D.F.: An algebraic formulation of Knuthian semantics. 17th IEEE Symp. on FOCS, 1977, 127-136.

[DRT81] Demers, A., Reps, T., Teitelbaum, T.: Incremental evaluation for attribute grammars with application to syntax-directed editors. POPL 8, 1981, 105-116.

[EnF81] Engelfriet, J., File, G.: Passes, sweeps, and visits. Lecture Notes in Comp. Sci. 115, Springer 1981, 193-207.

[Gan80] Ganzinger, H.: Transforming denotational semantics into practical attribute grammars. Lecture Notes in Comp. Sci. 94 (1980), 1-64.

[Gie79] Giegerich, R.: Introduction to the compiler generation system MUG2. Report TUM-INFO-7913, Techn. Univ. München, 1979.

[Gie82] Giegerich, R.: Automatic generation of machine-specific code optimizers. POPL 9, 1982, 75-81.

[GMW80] Glasner, I., Möncke, U., Wilhelm, R.: OPTRAN, a language for the specification of program transformations. Informatik-Fachberichte 34, Springer, March 1980, 125-142.

[GMW81] Giegerich, R., Möncke, U., Wilhelm, R.: Invariance of approximative semantics with respect to program transformations. Informatik- Fachberichte 50, Springer 1981, 1-10.

[GRW77] Ganzinger, H., Ripken, K., Wilhelm, R.: Automatic generation of optimizing multipass compilers. In: Gilchrist, B. (ed.): Information Processing 77, North-Holland Publ. Co., Amsterdam, New York, Oxford, 1977, 535-540.

[HeU75] Hecht, M.S., Ullman, J.D.: A simple algorithm for global data flow analysis programs. SIAM J. Comp. 4 (1975), 519-532.

[HLP78] Räihä, K.-J., Saarinen, M., Soisalon-Soininen, E. Tienari,M.: The compiler writing system HLP (Helsinki Language Processor). Dep't. of Comp. Science, Helsinki Univ., Report A-1978-2, 1978.

[JaW75] Jazayeri, M., Walter, K.G.: Alternating semantic evaluator, Proc. ACM Ann. Conf., 1975, 230-234.

[JaP81] Jazayeri, M., Pozefsky, D.: Space-efficient strorage management in an attribute evaluator. ACM TOPLAS 3, 4 (1981), 388-404.

[Joc81] Jochum G.: Automatische Konstruktion und einheitliche Darstellung von Attributauswertungsalgorithmen. TUM-I8113, Techn. Univ. München, June 1981.

[Joh75] Johnson, S.C.: YACC: yet another compiler compiler. Techn. Rep. CSTR32, Bell Labs, Murray Hill, 1975.

[JoS80] Jones, N.D., Schmidt, D.A.: Compiler generation from denotational semantics.

Lecture Notes in Comp. Sci. 94 (1980), 70-93.

[JoW81] Jochum, G., Willmertinger, W.: A tool for developing text processing systems: translator writing systems. Report TUM-I8103, Techn. Univ. München, 1981.

[KaU76] Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Informatica 7 (1977), 305-317.

[KeW76] Kennedy, K., Warren, S.K.: Automatic generation of efficient evaluators for attribute grammars. POPL 3, 1976.

[Kro75] Kron, H.H.: Tree templates and subtree transformational grammars. PhD-thesis, Univ. of Cal., Santa Cruz, 1975.

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

[PQC79] Leverett, B.W., Cattell, R.G.G., Hobbs, S.O., Newcomer, J.M., Reiner, A.H., Schatz, B.R., Wulf, W.A.: An overview of the production quality compiler-compiler project. Dept. of Comp. Science, Carnegie-Mellon University, CMU-CS-79-105, 1979.

[LRS73] Lewis, P.M., Rosenkrantz, D.J., Stearns, R.E.: Attributed translations. Proc. ACM 5th Annual Symp. on Theory of Comp., 1973.

[Mön82] Möncke, U.: Doctorial dissertation, Univ. Saarbrücken, forthcoming.

[Mos76] Mosses, P.: Compiler generation using denotational semantics. Proc. Symp. on Math. Found. of Comp. Sci., Lecture Notes in Comp. Sci., 45 (1976), 436-441.

[Mos79] Mosses, P.: SIS - Semantics implementation system. Reference Manual and user guide. Report DAIMI MD-30, Univ. Aarhus, 1979.

[Pau81] Paulson, L.: A semantics-directed compiler generator. POPL 9, 1982, 224-233.

[Poz79] Pozefsky, D.P.: Building efficient pass-oriented attribute grammar evaluators. Univ. North Carolina at Chapel Hill, UNC TR 79-006, 1979.

[Räi79] Räihä, K.-J.: Dynamic allocation of space for attribute instances in multipass evaluators of attribute grammars. Proc. SIGPLAN Symp. on Compiler Construction, Boulder, 1979, 26-38.

[Rep82] Reps, T.: Optimal time incremental semantic analysis for syntax-directed editors. POPL 9, 1982, 169-176.

[Ros77] Rosen, B.K.: High-level data flow analysis. CACM 20, 10 (1979), 712-724.

[Set81] Sethi, R.: Circular expressions: elimination of static environments. 8th ICALP, Lecture Notes in Comp. Sci. 115 (1981), 378-392.

[Tar76] Tarjan, R.E.: Iterative algorithms for global data flow analysis. Report STAN-CS-76-547, Stanford Univ., 1976.

[Wil79] Wilhelm, R.: Computation and use of data flow information in optimizing compilers. Acta Informatica 12 (1979), 209-225.