OPTRAN - A Language/System for the Specification of Program Transformations: System Overview and Experiences*

Peter Lipps, Ulrich Möncke, Reinhard Wilhelm

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

ABSTRACT

OPTRAN is a batch-oriented system for the generation of compilers that support program transformations. Programs are represented by attributed abstract syntax trees (AAST). The transformation of AAST's is a powerful method to describe problems in compiler writing such as machine-independent optimisations, language-based editors, and source-to-source translations.

The specification language OPTRAN allows for a static and declarative description of tree transformations. Given such a specification, the system will automatically generate the transformation system, mainly consisting of an attribute evaluator and reevaluator, as well as a tree analyzer and transformer.

The paper presents an introduction to the description mechanisms together with an overview of the system, showing the interaction of several generators. The main goal of the system design is the usage of precomputation methods wherever possible. This generative approach is explained. The static view of transformations makes it possible to generate highly efficient transformers but also has its limitations, which we mention.

The system is written in Pascal and generates Pascal programs. Pascal also serves as host language, i.e. semantic rules are specified as Pascal procedures. This complicates the error diagnosis of the runtime system as the semantics of these procedures is not obvious to the generator system. Furthermore it inhibits the recognition of certain properties of the specification like invariance of attribute assignments under non-trivial transformations.

We report practical experiences for some applications, e.g. a compiler for MiniPascal producing M68000 code and a frontend for Ada producing DIANA intermediate descriptions. [Li88] contains an annotated bibliography on OPTRAN.

*partially supported by the Deutsche Forschungsgemeinschaft under Project "Manipulation of Attributed Trees" and the Commission of the European Community under Esprit Project Ref. No. 390 (PROSPECTRA)

1. Introduction

Attributed abstract syntax trees (AASTs) have become a standard representation for programs in software development and compiling environments. Many tasks such as standardization, code optimization, source-to-source translation, and transformational program development can be specified by attributed tree transformations. The **OPTRAN specification language** allows the definition of sets of such tree transformation rules. The **OPTRAN system** consists of a set of generators producing efficient tree transformers from an OPTRAN specification.

We will now abstract from the applications listed above and discuss different views of the concept "tree transformation".

The Term rewriting view

Transformations of attributed trees can be seen as being derived from term rewriting: There, transformations are specified by a set of simple transformation rules (term rewriting rules, oriented equations). The rules are specified independently and not linked together by any superimposed control structure. Rules may be applied ("fired") whereever their left sides match inside the object tree. Rule application causes local changes in an object tree, which represents the state of the transformation system. OPTRAN is based on this batch oriented pure term rewriting scheme with the following extensions:

- rule application may depend on attribute values
- rule application may modify attribute values
- rule application conflicts are solved by user defined strategies mainly concerning the orientation in the object tree (bottom-up, top-down, left-to-right, right-to-left).

The enrichment of term rewriting by attributes leads to the problem of attribute updating and ensuring attribute consistency, which is solved by reevaluation techniques. The advantage of this approach is the locality both of rule and attribute specification, the latter locally to a (operator resp.) production. On the other hand, this scheme lacks powerful constructs for describing strategies, linking rules together, building rule packages, coupling application of different rules, conditional and (locally) iterative rule application, and specifying tree walking. This deficit of describing control was recognized early. Proposals for a PASCAL like control structure have been made, see for example [GMW80] . In addition, proposals for language extensions like list constructs [Ba86] and their attribution [Ra86, BMR85] as well as more powerful patterns [BMR86] exist. [Th88] investigates coupling of transformation units and the maintenance of transformation rule packages.

The Functional view

Transformations are not specified as a set of unrelated rules, but as functions to be applied to the subject tree. The style of describing control is very different from the term rewriting or the OPTRAN style. Functional composition may be used to form bigger transformations from smaller ones. The simple patterns and rules mentioned above do not vanish, but are embedded as parts of the functional expression describing the whole transformation. Higher order functions may be used to formulate transformation strategies. Nevertheless, the experience from the construction of pattern matching automata etc. may be amortised in the implementation of such a functional language. The reader is referred to [He88] for more details.

System overview

OPTRAN is a very-high-level language for the specification of attributed abstract syntax trees and tree transformations. The structure of an abstract syntax tree is described by a regular tree grammar. A set of attributes is associated with each node of a tree. Context information may be collected in attributes and passed over the tree. There is a set of semantic rules for each production of the tree grammar describing the functional dependencies between attributes. Transformation rules specify the possible modifications of trees. Each rule consists of an application condition (left hand side) and an output description (right hand side). The syntactic part of an application condition is described by an input pattern. An additional context condition can be formulated by means of a predicate over attributes of the input pattern. A transformation rule is said to be applicable at a node of a tree if there is a match for the input pattern at that node and the predicate is satisfied. If a rule is a applicable the part of the tree

that is matched by the input pattern is replaced by the output pattern. Hereby, the bindings of the variables of the input template resulting from the successful match are used to construct the new tree. In general, the transformation of a tree changes the functional dependencies of attributes. Therefore, the values of inconsistent attributes, i.e. attributes for which the dependencies have been changed by a transformation, have to be recomputed.

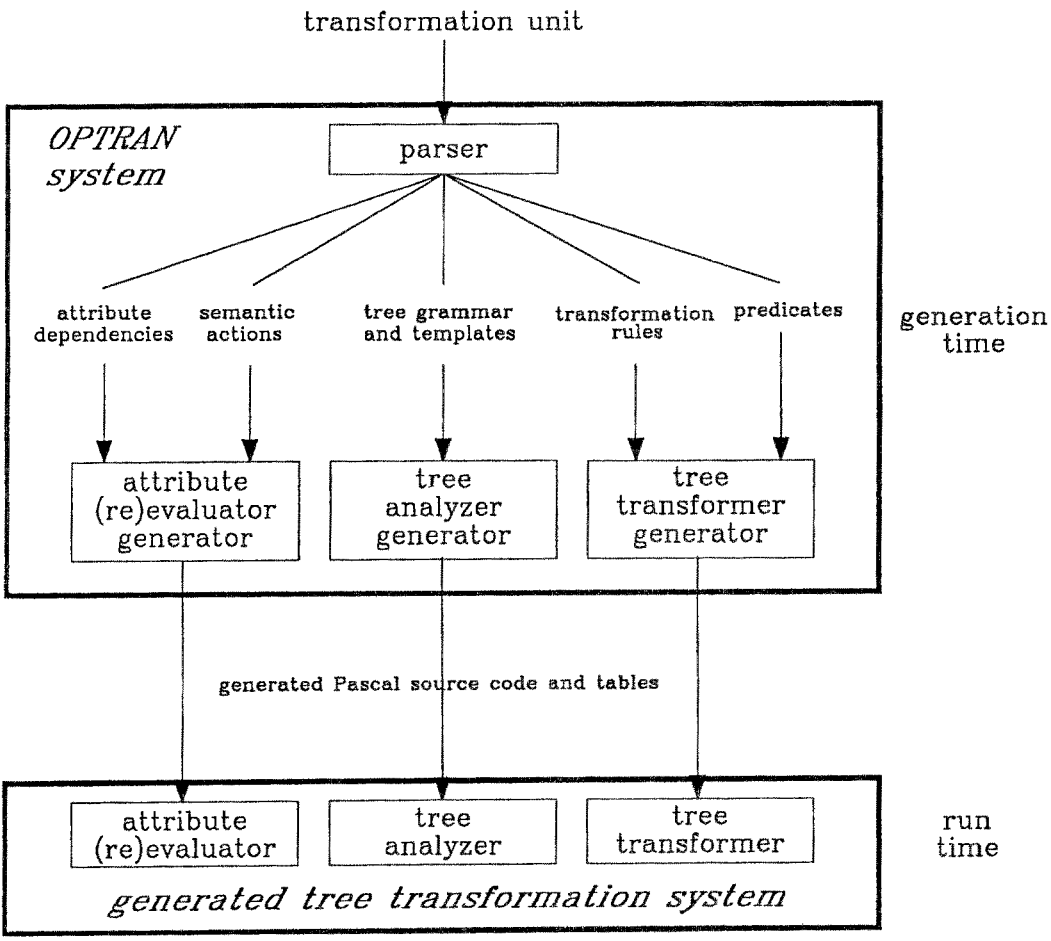

Figure 1

An OPTRAN specification of a tree transformation system is called a *transformation unit (t-unit)*. A t-unit mainly consists of three parts, i.e.

- a tree grammar, describing the structure of abstract syntax trees,
- attributes associated with nodes of the tree and semantic rules associated with productions denoting the functional dependencies between attributes,
- a set of tree transformation rules.

The OPTRAN system generates a *transformation system (t-system)* for a given t-unit. At run time, the t-system takes an abstract syntax tree and basically performs the following actions:

- initially, the *attribute evaluator* computes the value of all attributes according to the functional dependencies specified in the t-unit
- the *tree analyzer* searches the attributed abstract syntax tree for a node where a tree template matches and the corresponding predicate is satisfied;

- the *tree transformer* applies the selected transformation rule to the tree;
- the *attribute reevaluator* recomputes the value of inconsistent attributes of the transformed tree.

Extensive analysis of the static properties of a t-unit at generation time makes it possible to perform these tasks in a very efficient way.

- the *attribute (re)evaluator generator* analyses the attribute dependencies
- the *tree analyzer generator* analyzes the set of input patterns of the transformation rules and produces an efficient tree pattern matcher from it [We83] ;
- the *tree transformer generator* generates efficient tree transforming programs for the transformation rules [Wi81] ;

Figure 1 shows the basic components of the OPTRAN system and how they are connected.

2. Specification language

Abstract syntax trees are described by a regular tree grammar in OPTRAN. The grammar corresponds to the tree part of a string-to-tree grammar and makes linking to a compiler frontend possible. At this time, such a frontend can be generated with the POCO LALR(1)-Parser-Generator [Eu88] or the ELL(2)-Parser-Generator [He86a, He86b] both developed at the Universität des Saarlandes.

A *tree grammar* is a quadrupel $TG = (N, OP, P, S)$, where N is a finite set of *nonterminals*, OP is a finite set of *operators* with fixed arity, $S \in N$ is the *axiom*, and P is a set of *productions* of the form $X_0 ::= X_1$ or $X_0 ::= \langle op, X_1, ..., X_n \rangle$, where $n \in N_0$ is the *arity* of the operator *op*, $X_0, \cdots, X_n$ are nonterminals, and $\langle op, X_1, ..., X_n \rangle$ is the linear representation of a tree with depth 1. An operator *op* must not appear in more than one production.

An *abstract syntax tree t* of *TG* is a finite ordered tree. The nodes of *t* are labeled with operators.

Example:

```
STATLIST ::=   <sepop, STATLIST, STAT>
STATLIST ::=   STAT
STAT       ::=   ASSIGNSTAT
STAT       ::=   WHILESTAT
STAT       ::=   IFSTAT
ASSIGNSTAT ::=   <assop, VARIABLE, EXPR>
WHILESTAT  ::=   <whileop, EXPR, STATLIST>
IFSTAT     ::=   <ifop, EXPR, STATLIST, STATLIST\2>
EXPR       ::=   <addop, EXPR\1, EXPR\2>
EXPR       ::=   <relop, EXPR\1, EXPR\2>
EXPR       ::=   <intconst>
EXPR       ::=   <boolconst>
EXPR       ::=   VARIABLE
VARIABLE   ::=   <varid>
```

Figure 2

Attributes are used to collect context information and to spread this information over the tree. Each attribute has a type. Two disjoint sets of attributes, i.e. the set of *inherited* attributes *Inh(op)* and the set of *synthesized* attributes *Syn(op)*, are associated with each operator *op*. The set of all attributes of *op* is denoted by $Attr(op) = Inh(op) \cup Syn(op)$. *Instances* of these attributes are attached to each node of a syntax tree labeled with *op*.

A (possibly empty) set of *semantic rules* is specified for each production $X_0 ::= \langle op, X_1, \cdots, X_n \rangle$, describing the functional dependencies between attribute occurrences of this production. We call the occurrences of the inherited attributes at *op* and the synthesized attributes at X_i *used attribute occurences* and the the occurrences of the synthesized attributes at *op* and the inherited attributes at X_i *defined attribute occurences*, where $i \in \{1, ..., n\}$. There is exactly one *implicit* semantic rule *r* for each defined attribute occurrence of a production, that is not imported. Each used attribute occurrence can be an argument of *r*. Some defined attribute occurrences do not have an associated defining rule. The

corresponding instances will initially receive their value by importing it from a preceding frontend. These attribute occurrences are called *imported.*

The set of transformation rules specifies how to manipulate a given tree. A *transformation rule tr* consists of an *application condition* and an *output description.* The application condition has a syntactic and a semantic part. An input pattern specifies the *syntactic application condition.* A leaf of a pattern may be labeled with a *parameter.* A parameter matches every tree. Input patterns have to be linear, that is no parameter appears twice in the pattern. The semantic application condition is given by a *predicate* over the attributes of the input pattern. A rule *tr* is **applicable** at a node *n* of a tree *t* if the input pattern matches at *n* and the predicate over attribute instances of the subtree at *n* is satisfied. *n* is called *transformation node.* The predicate can be omitted.

Application of a transformation rule *tr* restructures the tree: the instance of the matching input pattern is replaced by the corresponding output pattern and each parameter of the output pattern is replaced by the matched subtree. The value of an imported attribute is specified by a so called *explicit* semantic rule, which may take any attributes of the input pattern as argument. The value of the non-imported attributes is defined by the implicit semantic rules specified for each production of the tree grammar.

Example:

```
transform
      <ifop, <boolconst>, THENPART, ELSEPART>
if istrue(svalue of boolconst) into
      THENPART
else into
      ELSEPART
fi;

transform
      <varid>
if (sconst of varid) and (stype of varid = inttype) into
      <intconst>
elsif (sconst of varid) and (stype of varid = booltype) into
      <boolconst>
fi;
```

Figure 3

In general, there may be more than one transformation rule applicable in a tree at the same time. A user defined *strategy* [We83] resolves the conflict: the user can specify if a given tree is searched for rule application *bottom-up* or *top-down*, *left-to-right* or *right-to-left*. In addition, a (*strictly*) *monotonic* strategy guarentees that only (proper) ancestors/descendants of a transformation node are tested for rule application. The rule with the most specific input pattern is chosen if there is more than one transformation rule applicable at the same node. If two patterns are incomparable the textually preceding rule is chosen.

An attributed tree grammar, a set of transformation rules and the specification of a strategy for conflict resolution together constitute a *transformation unit.*

3. Attribute (re)evaluation

3.1. Generation time analysis

The analysis of static properties of a given transformation unit at generation time renders it possible to generate very efficient transformation systems. The generative approach to tree pattern matching is explained in [MWW86] , where it is compared to an interpretative approach. In this paper, we only give a short overview of the generative aspects of attribute (re)evaluation. A detailed description of

attribute (re)evaluation in OPTRAN can be found in [LMOW87] .

Using a pure interpretaive approach an attribute (re)evaluator would at run time follow the attribute dependencies in a given tree. The generated attribute (re)evaluator uses precomputed information about attribute dependencies. The precomputation is based on the analysis of the tree grammar at generation time. In the OPTRAN system it is performed using *grammar flow analysis (GFA)* [MW83, Mö85, Mö86a] . GFA simulates the transformation time behaviour at generation time by computing fixed points in a graph structure, called *grammar graph* [MW83] , that represents the tree grammar. Depending on the actual *grammar flow problem*, the precomputed information can be exact or approximative. At run time, this information is associated with the nodes of the actual abstract syntax tree.

The computation of characteristic graphs, ordered partitions of attributes, attribute evaluation plans [Ol86] and identity classes [Ti86] can be formulated as grammar flow problems. In addition, grammar flow analysis can be used to generate tree pattern matching automata [Mö86a] and code generators [Mö87] .

The generative support for attribute (re)evaluation is motivated in several ways. Utilization of characteristic graphs makes reevaluation in time O(|number of changed attribute instances|) feasible [Re82] . The set of characteristic graphs resp. the IO-graph [KW76] can be precomputed for each nonterminal. At run time, there is no need to propagate graphs in the actual tree or to build transitive closures. Instead, only encodings of graphs are propagated and an automaton selects the right graph from the set of graphs at a nonterminal, depending on the encodings of graphs at the children. In addition, the partition of attribute instances into classes of simultaneously evaluable instances together with a total order on these classes can be preplanned. Furthermore, the automata that perform the selection process at run time can be compressed at generation time [BMW87] .

OPTRAN can handle absolutely noncircular attribute grammars, if the IO-graphs, i.e. approximative subordinate characteristic graphs, are precomputed. A pass evaluation scheme can be incorporated [LMOW87] .

3.2. Language features

A detailed survey of the following language features and a description of their implementation can be found in [GPSW86, LMOW87] .

Demand and data driven reevaluation

The OPTRAN system offers a *data driven* and a *demand driven* attribute (re)evaluator. Both evaluation mechanisms can be freely mixed. Using the data driven scheme, all attributes that may have changed their value are recomputed, i.e. a new consistent attribute value for an attribute instance is computed if at least one argument of the semantic rule defining the value of this instance has received a new value. The demand driven scheme allows to delay the reevaluation of attributes until their values are needed.

Complex attributes and footholds

When using the demand driven scheme attributes can be declared *complex*. Utilization of complex attributes instead of so-called simple attributes leads to smaller memory requirements since the value of a complex attribute instance is not stored. To speed up the reevaluation process some complex attributes can be made *footholds* (actually denoted *based* attributes): the value of a foothold is stored permanently in the tree.

Local attributes

Local attributes associated with a production rather than a nonterminal [Mö86b] can be used to store some temporary result of a computation that is used by more than one semantic rule.

Identity classes

Attribute instances identically depending on each other constitute an *identity class* and share the same memory allocation. An application of a transformation rule can modify identity classes.

4. Host and implementation language

Pascal is used as *implementation language* and as *host language*, i.e. the OPTRAN system itself is written in Pascal, semantic rules have to be coded as Pascal procedures/functions and attribute types are Pascal data types. In the following we will comment on both design decisions. When judging these descisions, one has to keep in mind that the developement of OPTRAN started in the late 70's. A first system including all parts described in section 1 except an attribute reevaluator was available in 1982 [SSW82, MWW84] .

Implementation language

The source of all generators makes up more than 39.000 lines of code and the static part of a run time transformation system has nearly 14.000 lines. The following deficiencies of Pascal made the implementation and maintenance of the OPTRAN system particularly hard, as will be the case for any system of comparable size:

- lack of modularity
  During the development of OPTRAN, there were more than 13 programmers involved in the implementation of the system. Even though the system itself consists of several modules, some modules had to be maintained by more than one programmer at the same time. Here, we suffered from the absence of an accepted module concept in Pascal.
- under-developed operating system interface
  The system's resource management (file system, memory manager) could have been optimized if there was better access to OS services than those offered by standard Pascal (e.g. no random file access). Using non-standard features of some Pascal dialects is no solution since it makes the OPTRAN system non-portable.
- strong type checking
  The strong type checking of Pascal has to be circumvented using Pascal variant records leading to somewhat unnatural formulation of otherwise (e.g. in C) fairly obvious parts. In addition, some (generic) operations had to be recoded for several data types, leading to a great amount of code duplication. A less rigid type system, generic procedures or polymorphic functions would have solved this problem;
- no separate compilation
  The fact that a lot of the system's modules are fixed or won't be changed after their generation can not be exploited by the system, i.e. changing a single semantic rule forces one to recompile the whole run time transformation system. Berkeley Pascal offers seperate compilation, but again this is a non-standard feature.

Making a UNIX host (SUN 3/160 running SUN-OS 3.4) our development machine the situation has been improved. The usage of the source code control system (SCCS) and the make utility facilitated the maintenance of the system and made it possible to reflect the modularity of the system in a reasonable way. However, the modularity of the generated transformation system is only exploited with Berkeley Pascal's seperate compilation feature.

The run time system still suffers from a few specific Pascal limitations. Generated static tables can not be included in the Pascal source but have to be read from a file during the initialisation phase of the transformation system. Interfacing a frontend, i.e. reading an abstract syntax tree, is done via file i/o, too, since Pascal does not support input/output of dynamic data structures. Practical experiences reveal that the system's run time is dominated by the time spent for i/o.

The run time library of some Pascal compilers is another cause for unsatisfying run times. Automatic translation of some system components from Pascal to C cut down the generator run time significantly.

We have successfully ported the system to a Siemens 7.5xx running BS2000. Unfortunately, we encountered several Pascal compiler errors and limitations (e.g. incorrect code generation, static internal symbol tables) during this port and an attempt to bring it to another UNIX system. Especially, the unusual size (> 64 KBytes) of data structures used in some part of the OPTRAN system lead to several problems.

Host language

Attribute types are specified as Pascal data types (e.g. integer, record) and semantic rules are

written as Pascal procedures/functions. In addition, one has to specify special compare and copy/dispose operations for each attribute type which is used internally by the system at run time. The behaviour of the system can be manipulated by changing these operations. Such a modification can be intended, e.g. to artificially restrict the region of reevaluation in the transformed tree or to make access to context information more efficient. On the other hand, these modifications are the source of errors that are hard to find since it is not easy to distinguish them from possible bugs of the system itself.

Furthermore, users of the system misuse side effects of Pascal routines to manipulate global data structures. This again may lead to hard-to-locate errors, especially because the user's expectation of system behaviour often differs from what is really going on inside the system.

Besides, the possibility of side effects seriously interferes with the analysis of a transformation's semantics at generation time. In general, the generating system has no knowledge of the effect of a given semantic rule. Actually, there is one exception to this statement: identity functions (called copy rules in [Ho86]) are treated in a special way by the system.

Using a functional language as host language would - at least in some cases - make it possible to reason about safeness and invariance of attribute values with regard to application of transformations [GMW81] . There is a trade-off between an improvement of generation time analysis that can be achieved by the usage of a functional language and the exploitation of side effects in an imperative language, e.g. manipulation of a global state. However, taking into account the inherent functional nature of attribute grammars we plead for a functional host language.

5. Practical experience

Code optimisation and data flow analysis

OPTRAN has proven its usefulness in classical fields of compiler construction, i.e. machine-independent code optimisation and data flow analysis [LMOW87] . [Li86] contains a transformation unit for a toy language called BLAN. A fraction of this t-unit is shown in the appendix. The generated t-system performs static semantic checking, global data flow analysis and constant propagation. The attribution for global data flow analysis and the transformation rules for constant propagation are based on algorithms presented in [Wi79a] . For this application the power of OPTRAN tree patterns is sufficient. The mixture of demand and data driven attributes accelerates the (re)evaluation process. Automatic reevaluation is fully utilized since most of the transformation rules affect the set of constant variables, which is represented by a special demand attribute. In addition, this attribute is an argument of some predicates. As stated in [Wi79a] , non-circular attribute grammars compute good approximations to the exact data flow information. The class of absolutely noncircular attribute grammars which can be handled by OPTRAN is sufficient for BLAN. Currently, [Ti88] investigates the effect of circular grammars into the specification language and the generation mechanism. A simple way of simulating circular attribute grammars by transformations and attribute reevaluation is described inTh88 .

Ada→DIANA frontend

The Ada DIANA frontend produces an intermediate description in DIANA for a given Ada program. The specification of this frontend consists of two t-units.

The transformation system that is generated for the first t-unit [Ke88] produces a normalized OPTRAN tree, where general operators are replaced by specific operators after name class analysis, e.g. the tree for an Ada construct like *f(a)* is replaced by a subtree that represents either a function call or an array access. The actual choice is dependent on the context, which is accumulated in attributes. These attributes serve as arguments for predicates, thereby driving the transformation process. The replacement of subtrees itself is described by the syntactic part of the transformation rules. The specification for this t-unit (14300 lines) contains 116 transformation rules, 569 productions and 4211 semantic rules.

The second t-unit (15000 lines) [Ma88] mainly specifies overload resolution. It contains no transformation rules. The generated run time system is merely used as an attribute evaluator. The specification contains 550 productions and 4950 semantic rules.

The frontend itself will be described in [KM88] . Both t-units do not make use of the full power

of the OPTRAN system. As said before, the second one even does not contain any transformations. Therefore, there is no need for a sophisticated reevaluation process. On the other hand, the evaluator is designed with transformations in mind: special memory management techniques that are only useful in "pure" attribute evaluators [FY86] are not exploited. In the first t-unit context information is collected in symbol table attributes during initial attribute evaluation. The transformations are guaranteed to leave this information unchanged. Unfortunately, there is no explicit way to specify invariance of attributes with regard to a transformation. However, there is an implicit way, i.e. modifying the corresponding compare function, as described in section 4. Using a kind of global attribute is another dangerous possibility. [Li86] exhibits the special risks of using global attributes in transformation systems.

Code generator for Pascal

An early version of the OPTRAN system without reevaluation (the reevaluation process was replaced by several runs of the initial attribute evaluator) was used to generate Motorola 68000 machine code for a subset of Pascal [Ra86] . Code emission is done by explicit rules when replacing Pascal tree fragments with special register nodes. Semantic rules describe register allocation and assignment. The assignment is computed during initial evaluation and is protected against reevaluation. The emphasis lays more on efficient tree transformation than on clever reevaluation. The generated t-system was modified by hand to make attribute values dependent on the transformation history. However, the OPTRAN philosophy does not support this view: in OPTRAN, the value of an attribute instance is solely defined by the functional attribute dependencies.

At the time of writing the specification is adapted to the current system.

Other applications

A source-to-source translator, that translates SPL4 (Siemens system programming language) programs into C programs is part of [Ho88] .

These partly nontrivial examples show, that OPTRAN can be successfully used for complex translation and transformation tasks. It, however, requires a disciplined specifier, which withstands the temptation to gain efficiency by using all the possibilities Pascal as a specification language offers him.

6. Acknowledgements

Over the years, the following persons were involved in the design and/or implementation of the OPTRAN system and/or wrote specifications in OPTRAN: Peter Badt, Jürgen Börstler, Michael Greim, Bernd Edelmann, Stefan Edelmann, Thomas Geiger, Günther Horsch, Paul Keller, Peter Lipps, Thomas Maas, Ulrich Möncke, Matthias Olk, Stefan Pistorius, Peter Raber, Thomas Rauber, Peter Schley, Michael Schmigalla, Alois Schütte, Monika Solsbacher, Winfried Thome, Elisabeth Tieser, Heiner Tittelbach, Beatrix Weisgerber, and Reinhard Wilhelm.

Appendix A

```
{  1} t-unit B_LAN;
{  2} (* Authors:      Peter Lipps, Matthias Olk      *)
     ...
{ 17} (*  constant declaration part      *)
{ 18} const
     ...
{ 22}   maxvector    =  127;
{ 23}   maxnesting   =    4;
     ...
{ 160} (*  attribute type declaration part*)
{ 161} simpletype mputype
{ 162}     (* user defined type for mod, pre, use *)
{ 163}   veclen  = 1..maxvector;
{ 164}   bitvector       = set of veclen;
{ 165}   bvindex = 0..maxnesting;
{ 166}   bvarray = array [bvindex] of bitvector;
{ 167}   mputype         = bvarray;
```

```
{ 168}   operations
{ 169}     compare
{ 170}       function mputypecomp (mpu1, mpu2 : mputype) : boolean;
{ 171}     copy
{ 172}       procedure mputypecopy (mpu1 : mputype; var mpu2 : mputype);
{ 173}  endtype
     ...
{ 295}  (*  attribute declaration part      *)
{ 296}  (*          synthesized attributes   *)
{ 297}  synthesized
     ...
{ 324}      spre     : mputype attached to
{ 325}      (* preserved variables *)
{ 326}      parid, nopar, parlistop, procop, whileop, ifop, assop, sepop;
     ...
{ 348}  (*          inherited attributes     *)
{ 349}  inherited
     ...
{ 365}      ipool       : symtabtype attached to
{ 366}      (* constant pool *)
{ 367}      sepop, assop, ifop, whileop, procop, relop, addop, intconst, boolconst, varid;
     ...
{ 384}  (*  declaration of semantic rules *)
     ...
{ 515}  procedure unionmpu (bvarr1, bvarr2 : mputype; var bvarr3 : mputype);
{ 516}  procedure intersectmpu (bvarr1, bvarr2 : mputype; var bvarr3 : mputype);
     ...
{ 591}  procedure searchinpool (pool : symtabtype; scan : scanattrtyp; var ausscan : scanattrtyp);
     ...
{ 622}  (*  productions of attributed grammar  *)
{ 623}  grammar is
{ 624}
{ 626}  prog ::=  <op_prog, block>;
     ...
{ 697}  block ::= <op_block, blockhead, declpart, applpart>;
     ...
{ 740}  applpart ::=      <applpartop, stats>;
     ...
{ 744}    (* constant pool *)
{ 745}      create(solstruct, true, ipool of stats);
     ...
{ 747}  applpart ::=      <noapplpart>;
     ...
{ 751}  stats ::=  <sepop, stats\1, stats\2>;
{ 752}    local luse : mputype;
{ 753}    based spre of stats\1;
     ...
{ 757}    (* mod, pre, use *)
{ 758}      intersectmpu (suse of stats\2, spre of stats\1, luse);
{ 759}      unionmpu     (suse of stats\1, luse, suse of sepop);
{ 760}      intersectmpu (spre of stats\1, spre of stats\2, spre of sepop);
{ 761}      unionmpu     (smod of stats\1, smod of stats\2, smod of sepop);
{ 762}    (* constant pool *)
{ 763}      ipool of stats\1 := id(ipool of sepop);
{ 764}      ipool of stats\2 := id(spool of stats\1);
{ 765}      spool of sepop   := id(spool of stats\2);
{ 767}  stats ::=  stat;
{ 768}
{ 770}  stat ::=    callstat;
```

```
{ 772} stat ::=   assstat;
{ 774} stat ::=   whilestat;
{ 776} stat ::=   ifstat;
    ...
{ 780} assstat ::=      <assop, assvariable, expr>;
    ...
{ 788}   (* mod, pre, use *)
{ 789}     suse of assop := id(suse of expr);
{ 790}     initto0     (selement of assvariable, spre of assop);
{ 791}     initto1 (selement of assvariable, smod of assop);
{ 792}   (* constant pool *)
{ 793}     ipool of expr := id(ipool of assop);
{ 794}     spool of assop := insertinpool (sconst of expr, scvalue of expr,
{ 795}                         sscanattr of assvariable, selement of assvariable, ipool of assop);
    ...
{ 801} ifstat ::= <ifop, expr, stats\1, stats\2>;
{ 802}   local luse : mputype;
    ...
{ 810}   (* mod, pre, use *)
{ 811}     unionmpu (suse of stats\1, suse of stats\2, luse);
{ 812}     unionmpu (suse of expr, luse, suse of ifop);
{ 813}     unionmpu (spre of stats\1, spre of stats\2, spre of ifop);
{ 814}     unionmpu (smod of stats\1, smod of stats\2, smod of ifop);
{ 815}   (* constant pool *)
{ 816}     ipool of expr := id(ipool of ifop);
{ 817}     ipool of stats\1 := id(ipool of ifop);
{ 818}     ipool of stats\2 := id(ipool of ifop);
{ 819}     spool of ifop := intersect(equalpoolkey, lesskey, getobjkey,
{ 820}                         spool of stats\1, spool of stats\2);
    ...
{ 824} whilestat ::=    <whileop, expr, stats>;
    ...
{ 831}   (* mod, pre, use *)
{ 832}     unionmpu (suse of expr, suse of stats, suse of whileop);
{ 833}     allmpu   (spre of whileop);
{ 834}     smod of whileop  := id(smod of stats);
{ 835}   (* constant pool *)
{ 836}     ipool of expr := intersect (equalpoolkey, lesskey, getobjkey,
{ 837}                         ipool of whileop, spool of stats);
{ 838}     ipool of stats := minuspool (equalkey, lesskey, getobjkey,
{ 839}                         ipool of whileop, smod of stats);
{ 840}     spool of whileop := intersect (equalpoolkey, lesskey, getobjkey,
{ 841}                         ipool of whileop, spool of stats);
    ...
{ 916} expr ::=  simpleexpr;
    ...
{ 939} simpleexpr ::=  <addop, simpleexpr, simpleopd>;
    ...
{ 947}   (* mod, pre, use *)
{ 948}     unionmpu  (suse of simpleexpr, suse of simpleopd, suse of addop);
{ 949}   (* constant propagation *)
{ 950}     sconst of addop := sconst of simpleexpr and sconst of simpleopd;
{ 951}     addcvalue(sconst of simpleexpr, sconst of simpleopd,
{ 952}             scvalue of simpleexpr, scvalue of simpleopd, scvalue of addcp);
{ 953}   (* constant pool *)
{ 954}     ipool of simpleexpr := id(ipool of addop);
{ 955}     ipool of simpleopd := id(ipool of addop);
{ 957} simpleexpr ::=  simpleopd;
    ...
```

```
{ 961} simpleopd ::=  variable;
{ 963} simpleopd ::=  <intconst>;
{ 964}   local scanattr : scanattrtyp;
{ 965}   imported scanattr (*of intconst*);
{ 966}     stype of intconst := inttype;
{ 967}     sscanattr of intconst  := id( scanattr (*of intconst*));
{ 968}   (* mod, pre, use *)
{ 969}     creatempu (suse of intconst);
{ 970}   (* constant propagation *)
{ 971}     sconst of intconst := true;
{ 972}     makecvalue(scanattr (*of intconst*), scvalue of intconst);
     ...
{ 987} variable ::=     <varid>;
     ...
{ 995}   (* mod, pre, use *)
{ 996}     initto1 (extractelemcode(info), suse of varid);
{ 997}   (* constant propagation *)
{ 998}     sconst of varid := isinpool2 (ipool of varid, scanattr);
{ 999}     makecvalue(scanattr (*of varid*), scvalue of varid);
     ...
{1022} (*  declaration of transformation rules  *)
{1023} transformation is
{1024} strategy bulr sm;
{1025}
{1026} (tr1) transform
{1027}          <varid>
{1028}       if sconst of varid and (stype of varid = inttype)
{1029}       into
{1030}          <intconst>
{1031}       apply
{1032}          searchinpool(ipool of varid, scanattr of varid, scanattr of intconst);
{1033}       elsif sconst of varid and (stype of varid = booltype)
{1034}       into
{1035}          <boolconst>
{1036}       apply
{1037}          searchinpool(ipool of varid, scanattr of varid, scanattr of boolconst)
{1038}       fi;
     ...
{1056} (tr4) transform
{1057}          <addop, <intconst\1>, <intconst\2>>
{1058}       into
{1059}          <intconst>
{1060}       apply
{1061}          trafoaddop(sscanattr of intconst\1, sscanattr of intconst\2, scanattr of intconst);
     ...
{1131} (tr11) transform
{1132}          <ifop, boolconst, thenpart, elsepart>
{1133}       if boolfalse(sscanattr of boolconst)
{1134}       into
{1135}          elsepart
{1136}       else into
{1137}          thenpart
{1138}       fi.
```

References

[BMR85] **Badt, P., P. Raber, and U. Möncke, Attributation Schemata for List-Structured Nodes,** ESPRIT: PROSPECTRA Project Report S.1.3-R-1.0, FB 10 - Informatik, Universität des Saarlandes, 1985.

[BMR86] Badt, P., P. Raber, and U. Möncke, **Specification of Recursive Patterns,** ESPRIT: PROSPECTRA Project Report S.1.6-SN-4.0, FB 10 - Informatik, Universität des Saarlandes, 1986.

[BMW87] Börstler, J., U. Möncke, and R. Wilhelm, **Table Compression for Tree Automata,** Aachener Informatik Berichte No. 87-12, RWTH Aachen, Fachgruppe Informatik, Aachen, 1987.

[Ba86] Badt, P., **OPTRAN - Ein System zur Generierung von Baumtransformatoren: Rekursive Schablonen und Listenkonstrukte,** Diploma Thesis (in german), FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1986.

[Eu88] Eulenstein, M., **POCO - Ein portables System zur Generierung portabler Compiler,** *Informatik Fachberichte* 164,1988.

[FY86] Farrow, R. and D. Yellin, **A Comparison of Storage Optimizations in Automatically-Generated Attribute Evaluators,** *Acta Informatica* 23(4),Springer-Verlag, 1986.

[GMW80] Glasner, I., U. Möncke, and R. Wilhelm, **OPTRAN - A Language for the Specification of Program Transformations,** pp. 125-142 in *Informatik Fachberichte, 7th GI Workshop on Programming Languages and Program Development,* Springer-Verlag, 1980.

[GMW81] Giegerich, R., U. Möncke, and R. Wilhelm, **Invariance of Approximative Semantics with respect to Program Transformations,** pp. 1-10 in *11. GI Fachtagung für Programmiersprachen und Programmentwicklung,Informatik Fachberichte 50,* , Springer-Verlag, 1981.

[GPSW86] Greim, M., St. Pistorius, M. Solsbacher, and B. Weisgerber, **POPSY and OPTRAN-Manual,** ESPRIT: PROSPECTRA-Project Report, S.1.6-R-3.0, Technical Report No. A 08/86, FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1986.

[He86a] Heckmann, R., **An efficient ELL(1)-Parser Generator,** *Acta Informatica* 23, pp. 127-148, 1986.

[He86b] Heckmann, R., **Manual for the ELL(2)-Parser Generator and Tree Generator Generator,** Technical Report No. A 05/86, FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1986.

[He88] Heckmann, R., **A Functional Language for the Specification of Complex Tree Transformations,** *Proceedings of the 2nd European Symposium on Programming (ESOP'88), LNCS 300,* , pp. 175-190, Springer-Verlag,

[Ho86] Hoover, R., **Dynamically Bypassing Copy Rule Chains in Attribute Grammars,** ACM Symposium on Principles of Programming Languages, January 1986.

[Ho88] Horsch, G., **Source-to-source-Transformationen: Eine Studie des Compiler-Generierenden Systems OPTRAN anhand eines SPL4→C-Compilers,** Diploma Thesis (in german), FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1988.

[KM88] Keller, P. and Th. Maas, **An OPTRAN generated Ada→DIANA frontend,** Technical Report (forthcoming), FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1988.

[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.

[Ke88] Keller, P., **Spezifikation und Implementierung eines Compilerfrontends für Ada: Deklarations- und Nameclassanalyse ,** Diploma Thesis (in german), FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1988.

[LMOW87] Lipps, P., M. Olk, U. Möncke, and R. Wilhelm, **Attribute (Re)evaluation in OPTRAN,** ESPRIT: PROSPECTRA Project Report S.1.3-R-5.0, FB 10 - Informatik, Universität des Saarlandes, 1987. (to be published in Acta Informatica)

[Li88] Lipps, P., **OPTRAN: A Language/System for the Specification of Program Transformations - Annotated Bibliography,** Internal Report, FB 10 - Informatik, Universität des Saarlandes, 1988.

[Li86] Lipps, P., **Komplexe Attribute - Mechanismen zur Verwaltung und Berechnung in einem baumtransformierenden System,** Diploma Thesis (in german), FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1986.

[MW83] Möncke, U. and R. Wilhelm, **Iterative algorithms on grammar graphs,** pp. 177-194 in *Proc. 8th Conference on Graphtheoretic Concepts in Computer Science*, ed. H.J. Schneider, H. Goettler, Hanser Verlag, München, 1983.

[MWW84] Möncke, U., B. Weisgerber, and R. Wilhelm, **How to implement a system for manipulation of attributed trees,** pp. 112-127 in *Informatik Fachberichte, 8th GI Workshop on Programming Languages and Program Development*, Springer-Verlag, Zürich, March 1984.

[MWW86] Möncke, U., B. Weisgerber, and R. Wilhelm, **Generative support for transformational programming,** pp. 511-527 in *ESPRIT: Status Report of Continuing Work*, Elsevier Sc., Brussels, 1986.

[Mö85] Möncke, U., **Generierung von Systemen zur Transformation attributierter Operatorbäume - Komponenten des Systems und Mechanismen der Generierung,** Ph.D. Thesis (in german), FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1985.

[Mö86a] Möncke, U., **Grammar Flow Analysis,** ESPRIT: PROSPECTRA-Project Report S.1.3-R-2.1, FB 10 - Informatik, Universität des Saarlandes, 1986. (submitted for publication)

[Mö86b] Möncke, U., **Production Local Attributes,** ESPRIT: PROSPECTRA-Project Study Notes, S.1.3-SN-3.0, FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1986.

[Mö87] Möncke, U., **Simulating Automata for Weighted Tree Reductions,** ESPRIT: PROSPECTRA-Project Report S.1.6-R-5.1 and Technical Report No. A 10/87, FB 10 - Informatik, Universität des Saarlandes, 1987.

[Ma88] Maas, T., **Spezifikation und Implementierung eines Compilerfrontends für Ada: Auflösung der Überladung und Erzeugung von DIANA,** Diploma Thesis (in german), FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1988.

[Ol86] Olk, M., **Generierung eines effizienten Attributschedulers für ein baumtransformierendes System,** Diploma Thesis (in german), FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1986.

[Ra86] Raber, P., **Listenkonstrukte und ihre Attributierung in einem System zur Generierung von Baumtransformatoren,** Diploma Thesis (in german), FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1986.

[Ra86] Rauber, Th., **Registerverteilung und Codeselektion für wechselnde Zielmaschinen,** Diploma Thesis (in german), FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1986.

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

[SSW82] Schmigalla, M., A. Schütte, and B. Weisgerber, **OPTRAN'82-Manual,** Internal Report, FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, December 1982.

[Th88] Thome, W., **Ein Konzept zur Bibliotheksverwaltung von Transformationseinheiten und deren Hintereinanderschaltung,** Diploma Thesis (in german), FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1988.

[Ti88] Tieser, E., **Attributauswertung für zyklische Attributierung ,** Diploma Thesis (in german), FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1988.

[Ti86] Tittelbach, H., **Effiziente Attributspeicherverwaltung für ein baumtransformierendes System,** Diploma Thesis (in german), FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1986.

[We83] Weisgerber, B., **Attributierte Transformationsgrammatiken: Die Baumanalyse und Untersuchungen zu Transformationsstrategien,** Diploma Thesis (in german), FB 10 - Informatik, Universität des Saarlandes, Saarbrücken, 1983.

[Wi79a] Wilhelm, R., **Computation and Use of Data Flow Information in Optimizing Compilers,** *Acta Informatica* **12,** pp. 209-225, Springer-Verlag, 1979.

[Wi81] Wilhelm, R., **A Modified Tree-To-Tree Correction Problem,** *Information Processing Letters* **12**(3), pp. 127-132, North-Holland, 1981.