Transformation of Attributed Trees Using Pattern Matching

J. Grosch

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

Transformation of Attributed Trees Using Pattern Matching

Josef Grosch

Aug. 28, 1991

Report No. 27

Gesellschaft für Mathematik und Datenverarbeitung mbH
Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1
D-7500 Karlsruhe

Transformation of Attributed Trees Using Pattern Matching

Josef Grosch
GMD Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1, D-7500 Karlsruhe, Germany
+721-662226
grosch@karlsruhe.gmd.de

Abstract This paper describes a tool for the transformation of attributed trees using pattern matching. The trees to be processed are defined by a formalism based on context-free grammars. Operations for trees such as composition and decomposition are provided. The approach can be characterized as an amalgamation of trees or terms including pattern matching, with recursion, attribute grammars, and imperative programming. Transformations can either modify the input trees or map them to arbitrary output. Possible applications are the various transformation tasks in compilers such as semantic analysis, optimization, or the generation of intermediate representations. The design goals have been to combine an expressive and high level technique for transformation with flexibility, efficiency, and practical usability. A reliable development style is supported by static typing and checks for the single assignment property of variables. We give some example transformations and describe the input language of our tool called *puma*. The relationship to similar work is discussed. Finally, experimental results are presented that demonstrate the efficiency of our approach.

Keywords transformation, attributed trees, pattern matching

1. Introduction

The transformation of trees using pattern matching becomes an accepted technique. Several tools have been constructed recently that follow this principle [CoP90, HeS91, LMW89, Vol91]. Tools for code generation successfully use the same technique, too [AGT89, ESL89]. We present a new tool called *puma* and its input language for the transformation and manipulation of attributed trees and graphs [Gro91a]. *Puma* stands for *p*attern *ma*tching and *u*nification. Its intended application areas are the various transformation tasks in a compiler operating on abstract syntax trees or arbitrary graph structures. This includes semantic analysis, optimization, intermediate code generation, source-to-source translation, and eventually machine code generation.

The trees that are subject to pattern matching are described by a formalism based on context-free grammars. The tree nodes may be associated with attributes of arbitrary types. Node types are used to specify the properties of tree nodes. An extension mechanism induces a subtype relation among the node types. Pattern matching is extended to handle subtypes and attributes, too. Operations for the composition and decomposition of trees are supported by a concise notation.

The building blocks for a transformation are recursive subroutines, classified as procedures, functions, and predicates, with an arbitrary number of input and output parameters. The bodies of the subroutines consist of rules which are made up of patterns, conditions, statements, and expressions. The first two components control the applicability of a rule. The statements determine what has to be done whenever a rule is applicable. The expressions provide values for the output parameters and the function result.

Static type checking with respect to trees is provided. Variables are declared implicitly and they are checked for the single assignment property. There is read and write access to attributes stored in the tree which allows the construction of attribute evaluators. In all places it is possible to escape to hand-written code which provides the power and flexibility of the imperative programming style.

The output of the generator is a source module in one of the target languages C or Modula-2. This module allows for easy integration and cooperation with other modules, either hand-written or generated ones. The pattern matching is local and considers a region at the top of the current subtree, only. It is implemented by direct code and therefore efficient.

Our approach can be regarded from several points of view: From the point of view of imperative programming it is an extension by statically typed, attributed trees, constructs for composition and decomposition, and pattern matching. From the point of view of logic programming it omits backtracking and restricts pattern matching to one of the two terms being a ground term. It adds attributes which are stored in the terms (trees), static typing, input and output modes for parameters, and an easy escape to imperative features. From the point of view of functional programming it offers the simple style of functional programming which has always been present in imperative languages having functions and recursion. It adds the pattern matching facility. From the point of view of attribute grammars it allows the specification of attribute evaluation with explicit control of the evaluation order or visit sequences. This eases the use of global attributes and gives full control on side-effects.

The intended use of this tool proceeds in three steps: First, a tree is constructed either by a parser, a previous transformation phase, or whatever is appropriate. Second, the attributes in the tree are evaluated either using an attribute grammar based tool, by a *puma* specified tree traversal and attribute computations, or by hand-written code. Third, the attributed tree is transformed or mapped to another data structure by a *puma* generated transformation module. These steps can be executed one after the other or more or less simultaneously. Besides trees, *puma* can handle attributed graphs as well, even cyclic ones. Of course the cycles have to be detected in order to avoid infinite loops. A possible solution uses attributes as marks for nodes already visited.

A transformer module can make use of attributes in the following ways: If attribute values have been computed by a preceding attribute evaluator and are accessed in read only mode then this corresponds to the three step model explained above. A *puma* generated module can also evaluate attributes on its own. A further possibility is that an attribute evaluator can call *puma* subroutines in order to compute attributes. This is especially of interest when attributes depend on tree-valued arguments.

The tool supports two classes of tree transformations: *mappings* and *modifications*. Tree mappings map an input tree to arbitrary output data. The input tree is accessed in read only mode and left unchanged. Tree *modifications* change a tree by e. g. computing and storing attributes at tree nodes or by changing the tree structure. In this case the tree data structure serves as input as well as output and it is accessed in read and write mode.

The first class covers applications like the generation of intermediate languages or machine code. Trees are mapped to arbitrary output like source code, assembly code, binary machine code, linearized intermediate languages like P-Code, or another tree structure. A further variant of mapping is to emit a sequence of procedure calls which are handled by an abstract data type.

The second class covers applications like semantic analysis or optimization. Trees are decorated with attribute values, properties of the trees corresponding to context conditions are checked, or trees are changed in order to reflect optimizing transformations.

Puma is part of the Karlsruhe Toolbox for Compiler Construction [GrE90]. In particular it cooperates with the generator for abstract syntax trees *ast* [Gro91b] and the attribute evaluator generator *ag* [Gro89]. The attributed trees are defined and managed by a module generated with *ast*. A second module generated by *puma* creates and handles these trees. This way all the powerful operations for trees and graphs provided by *ast* are available such as reader and writer procedures or the interactive browser. For sake of simplicity we will deviate from reality in this paper and treat the definition of the tree structure as part of *puma*.

The rest of this paper is organized as follows: Section 2 presents a few simple examples of how to describe transformations with *puma*. Section 3 describes the input language of the tool. Section 4 sketches the implementation of the generated transformer module. Section 5 compares our approach with related work. Section 6 presents experimental results. Section 7 contains concluding remarks.

2. Tree Transformation by Pattern Matching

The probably easiest way to get an impression of our approach can be obtained by having a look at a few introductory examples. We will use the abstract syntax of simple arithmetic expressions as input data structure. Besides a few intrinsic attributes describing e. g. the values of constants we use an attribute called *Type*. It describes the type of every subexpression. Its domain are trees, too. The tree definition based on a context-free grammar shown in Example 1 specifies the structure of expressions and types.

Example 1: Tree Definition

```
Expr            = Type <
   Plus         = Lop: Expr Rop: Expr .
   Minus        = Lop: Expr Rop: Expr .
   Const        = [Value] .
   Adr          = <
      Index     = Adr Expr .
      Select    = Adr [Ident: tIdent] .
      Ident     = [Ident: tIdent] .
   >.
>.
Type            = <
   Int          = .
   Real         = .
   Bool         = .
   Array        = [Lwb] [Upb] Type .
   Record       = Fields .
>.
Fields          = <
   NoField      = .
   Field        = [Ident: tIdent] Type Fields .
>.
```

The names before the character ’=’ can be regarded both as rule names or nonterminals. The possible right-hand sides for one nonterminal are enclosed in angle brackets ’<’ and ’>’. Non-tree valued attributes are enclosed in square brackets ’[’ and ’]’. The attributes *Lwb*, *Upb*, and *Value* are of the default type *int*. The attribute *Ident* is of the user-defined type *tIdent*. The tree-valued attribute *Type* of the rule named *Expr* is written like every other right-hand side nonterminal. The selector names *Lop* and *Rop* used in the rules called *Plus* and *Minus* allow symbolic access to right-hand side elements having the same type.

The association of an attribute such as *Type* with a nonterminal like *Expr* adds this attribute to every right-hand side belonging to this nonterminal. Hence the rules *Plus* and *Minus* have three right-hand side elements.

The subroutine specification presented in Example 2 describes part of a transformation of expression trees to P-Code [NAJ76]. The procedure *P_Code* performs a recursive tree traversal.

Example 2: Generation of P-Code

```
PROCEDURE P_Code (Tree)

Plus  (Int  (), Lop, Rop) :- P_Code (Lop); P_Code (Rop); Emit (ADDI); .
Plus  (Real (), Lop, Rop) :- P_Code (Lop); P_Code (Rop); Emit (ADDR); .
Minus (Int  (), Lop, Rop) :- P_Code (Lop); P_Code (Rop); Emit (SUBI); .
Minus (Real (), Lop, Rop) :- P_Code (Lop); P_Code (Rop); Emit (SUBR); .
```

Its body consists of a sequence of rules. Every rule is made up of a pattern and a list of statements separated by ':-'. Whenever a pattern matches against the input parameter of the subroutine, the associated statement list is executed. The parameter of the procedure *P_Code* is of type *Tree* which means that every tree according to the tree definition is a legal argument. The procedure *Emit* is supposed to be an external subroutine that performs output.

Example 3: Computation of Type Sizes

```
FUNCTION TypeSize ([Type, Fields]) int

Int  ()             RETURN 4 .
Real ()             RETURN 4 .
Bool ()             RETURN 1 .
Array (Lwb, Upb, T) RETURN (Upb - Lwb + 1) * TypeSize (T) .
Record (F)          RETURN TypeSize (F) .
Field (_, T, F)     RETURN TypeSize (T) + TypeSize (F) .
NoField ()          RETURN 0 .
```

The function *TypeSize* in Example 3 transforms or maps trees to integer values. It is defined to operate on trees of types *Type* and *Fields* and it returns a value of a type named *int*. Again this subroutine performs a recursive tree traversal. Instead of producing a side-effect like in the previous example it represents a pure functional mapping and demonstrates the arithmetic expression capabilities of *puma*. The character '_' denotes a so-called don't care pattern.

Example 4: Testing two Types for Compatibility

```
PREDICATE IsCompatible ([Type, Fields], [Type, Fields])

Int  ()              , Int  ()              .
Real ()              , Real ()              .
Bool ()              , Bool ()              .
Array (Lwb, Upb, T1), Array (Lwb, Upb, T2) :- IsCompatible (T1, T2); .
Record (F1)          , Record (F2)          :- IsCompatible (F1, F2); .
Field (_, T1, F1)    , Field (_, T2, F2)    :- IsCompatible (T1, T2);
                                               IsCompatible (F1, F2); .
```

Example 4 presents the predicate *IsCompatible* which can be seen as a boolean function. It operates on two parameters of types *Type* or *Fields*. Accordingly, every rule has two patterns for matching against the two arguments. The first three rules lead to a return value of true if both patterns match both arguments. The fourth rule returns true if the attributes *Lwb* and *Upb* of the two argument trees match (have the same value) and if the recursive call *IsCompatible* returns true for the two types *T1* and *T2* of the array elements. If none of the rules matches the predicate returns false.

The subroutine given in Example 5 has three input and one output parameter. It computes the result type of a binary expression which depends on the types of the operands and on the operator. The third pattern of every rule is enclosed in curly brackets '{ and '}'. This represents so-called target code which is more or less passed unchanged and unchecked to the generated module. In this case *opPlus* and *opMinus* are named constants encoding operators. Without the curly brackets they would be treated as pattern variables. The expression after the symbol '=>' describes the value of the output parameter in case of a successful match. It consists of a tree

Example 5: Procedure with Input and Output Parameters

```
PROCEDURE ResultType (Type, Type, Operator: int => Type)

Int  () , Int  () , { opPlus  } => Int  () .
Real () , Real () , { opPlus  } => Real () .
Int  () , Int  () , { opMinus } => Int  () .
Real () , Real () , { opMinus } => Real () .
```

constructor which creates a tree having one node.

3. Specification Language

The description of a tree transformation consists of the definition of attributed trees and a set of subroutines.

3.1. Tree Structure

The structure of attributed trees or (directed) graphs is specified by a formalism based on context-free grammars. However, we primarily use the terminology of trees and types, here. A tree consists of *nodes*. A node may be related to other nodes in a so-called *parent-child* relation. Then the first node is called a *parent* node and the latter nodes are called *child* nodes. Nodes without a parent node are usually called *root* nodes, nodes without children are called *leaf* nodes.

The structure and the properties of nodes are described by *node types*. Every node belongs to a node type. A specification of a tree describes a finite number of node types. A node type specifies the names of the child nodes and the associated node types as well as the names of the attributes and the associated attribute types.

Children are distinguished by *selector* names which have to be unambiguous within one node type. The children are of a certain node type. Example:

```
Plus      = Lop: Expr Rop: Expr .
Index     = Adr: Adr Expr: Expr .
```

The example introduces two node types called *Plus* and *Index*. A node of type *Plus* has two children which are selected by the names *Lop* and *Rop*. Both children are of the node type *Expr*. If a selector name is equal to the associated name of the node type it can be omitted. Therefore, the node type *Index* can be abbreviated as follows:

```
Index     = Adr Expr .
```

As well as children, every node type can specify an arbitrary number of *attributes* of arbitrary types. Like children, attributes are characterized by a selector name and a certain type. The descriptions of attributes are enclosed in brackets '[' and ']'. The attribute types are given by names taken from the target language. Missing attribute types are assumed to be *int* or *INTEGER* depending on the target language (C or Modula-2). Children and attributes can be given in any order (see Example 1).

To allow several alternatives for the types of children an *extension* mechanism is used. A node type may be associated with several other node types enclosed in angle brackets '<' and '>'. Then this node type is called *base* or *super* type and the associated types are called *derived* types or *subtypes*. A derived type can in turn be extended with no limitation of the nesting depth. The extension mechanism induces a subtype relation between node types denoted by the symbol $\subseteq$. This relation is transitive. Where a node of a certain node type is required, either a node of this node type or a node of a subtype thereof is legal.

In Example 1 *Expr* is a base type describing nodes with one child called *Type*. The node type *Expr* has four derived types called *Plus*, *Minus*, *Const*, and *Adr*. The node type *Adr* is in turn extended by three derived types called *Index*, *Select*, and *Ident*. Where a node of type *Expr* is required, all mentioned node types are legal. Where a node of type *Adr* is required, nodes of the types *Index*, *Select*, or *Ident* are legal. Where a node of type *Index* is required, nodes of type *Index* are legal, only. The subtype relation is the transitive and reflexive closure of: $Plus \subseteq Expr$, $Minus \subseteq Expr$, $Const \subseteq Expr$, $Adr \subseteq Expr$, $Index \subseteq Adr$, $Select \subseteq Adr$, $Ident \subseteq Adr$.

Besides extending the set of legal node types, the extension mechanism has the property of extending the children and attributes of the base type. The derived types possess the children and attributes of the base type. They may define additional children and attributes. In other words they *inherit* the structure of the base type. The selector names of all children and attributes in an extension hierarchy have to be distinct. The syntax has been designed this way in order to allow single inheritance, only.

In Example 1 nodes of type *Expr* have one child selected by the name *Type*. Nodes of type *Plus* have three children with the selector names *Type*, *Lop*, and *Rop*.

3.2. Subroutines

A set of subroutines constitutes the main building blocks of a transformation. Like in programming languages, subroutines are parameterized abstractions of statements or expressions. There are three kinds of subroutines:

procedure : a subroutine acting as a statement
function : a subroutine acting as an expression and returning a value
predicate : a boolean function

Subroutines are specified according to the following syntax:

```
Subroutine = Header { Rule }
Header     = PROCEDURE Ident ( [ Parameters ] [ => Parameters ] )
           | FUNCTION  Ident ( [ Parameters ] [ => Parameters ] ) Type
           | PREDICATE Ident ( [ Parameters ] [ => Parameters ] )
Parameters = [ Ident : ] Type { , [ Ident : ] Type }
```

A subroutine consists of a header and a sequence of rules. The header specifies the kind of the subroutine, its name, and its parameters. In case of a function, the type of the result value is added. Input and output parameters are separated by the symbol '=>'. It suffices to give the type of a parameter. A name for the formal parameter is optional.

3.3. Types

Types are either predefined in the target language like *int* and *INTEGER*, or user-defined like *MyType*, or they are tree types like *Expr*. A tree type is described by the name of a tree definition, a single node type, or a list of node types enclosed in brackets '[' and ']'. In case of ambiguities the latter two kinds may be qualified by preceding the name of the tree definition.

```
Type     = TreeType | UserType
TreeType = Ident | [ Ident . ] Ident | [ Ident . ] '[' Idents ']'
UserType = Ident
Idents   = Ident { , Ident }
```

3.4. Rules

A rule behaves like a branch in a case or switch statement. It consists of a list of patterns, a list of expressions, a return expression in case of a function, and a list of statements.

```
Rule     = [ Patterns ] [ => Exprs ] [ RETURN Expr ] :- { Statement ; } .
Patterns = Pattern { , Pattern }
Exprs    = Expr { , Expr }
```

The semantics of a rule is as follows: A rule may succeed or fail. It succeeds if all its patterns, statements, and expressions succeed – otherwise it fails. The patterns, statements, and expressions are checked for success in the following order: First, the patterns are checked from left to right. A pattern succeeds if it matches its corresponding input parameter as described below. Second, the statements are executed in sequence as long as they succeed. The success of statements is defined below. Third, the expressions are evaluated from left to right and their results are passed to the corresponding output parameters. In case of a function, additionally the expression after RETURN is evaluated and its result is returned as value of the function call. The success of expressions is defined below, too. If all elements of a rule succeed then the rule succeeds and the subroutine returns. If one element of a rule fails the process described above stops and causes the rule to fail. Then the next rule is tried. This search process continues until either a successful rule is found or the end of the list is reached. In the latter case the behaviour depends on the kind of the subroutine: A procedure does nothing, a predicate returns false, and a function signals a runtime error. There is one exception to this definition of the semantics which is explained later.

3.5. Patterns

A pattern describes the shape at the top or root of a subtree. A pattern can be a decomposition of a tree, the keyword NIL, a label or a variable, one of the don't care symbols '_' or '..', or an expression. A decomposition is written as a node type followed by a list of patterns in parenthesis '(' and ')'. It may be optionally preceded by a label.

```
Pattern = [ Ident : ] Ident ( [ Patterns ] ) | NIL | Ident | _ | .. | Expr
```

The match between a pattern and a value is defined recursively depending on the kind of the pattern:

A decomposition with a node type t matches a tree u with a root node of type s if s is a subtype of t ($s \subseteq t$) and all subpatterns of t match their corresponding subtrees or attributes of u. If the node type is preceded by a label l then a binding is established between l and u which defines the label l to refer to the tree u.

The first occurrence of a label l in a rule matches an arbitrary subtree or attribute value u. All further occurrences of the label l within patterns of this rule match a subtree or an attribute value v only if u is equal to v. The equality for trees is defined in the sense of structural equivalence. Two attributes are equal if they have the same values. A binding is established between l and u which defines the label l to refer to the value u. The label can be used later to access the associated value.

The pattern NIL matches the values NULL or NIL. The don't care symbol '_' matches one arbitrary subtree or attribute value. The don't care symbol '..' matches any number of arbitrary subtrees or attribute values. An expression matches a parameter or an attribute if both have the same values.

3.6. Expressions

Expressions denote the computation of values or the construction of trees. Binary and unary operations as well as calls of external functions are written as in the target language. Calls of *puma* functions and predicates distinguish between input and output arguments. The syntax for tree composition is similar to the syntax of patterns.

```
Expr = Ident ( [ Exprs ] ) | NIL | Ident | _ | ..
     | Ident ( [ Exprs ] [ => Patterns ] )
```

The semantics of the different kinds of expressions is as follows:

A node type creates a tree node and provides the children and attributes of this node with the values given in parenthesis. NIL represents the value NULL or NIL. A label refers to the expression it was bound to upon its definition.

A function or predicate call must be compatible with the corresponding definition in terms of the numbers of expressions and patterns as well as their types. A function call evaluates the expressions corresponding to input parameters, passes the results to the function, and executes the function. Upon return from the function the result value of the function determines the result of this expression. The values of the output parameters that the function returns are matched against the actual patterns of the function call. If one pair does not match the call fails. Labels in the patterns may establish bindings that enable to refer to the output parameters or subtrees thereof.

The don't care symbols specify that no computation should be executed, either for one or for several expressions. The result values are undefined. The binary and unary operators (prefix and postfix) of the target language as well as array indexing, parentheses, numbers, and strings are known to *puma*. They are passed unchanged to the output.

In case of node types, labels for tree values, and functions returning tree values, *puma* does type checking. For user types, target code expressions, or target operators no type checking is done by *puma* but later by the compiler. An expression that does not contain calls of *puma* functions or predicates always succeeds. An expression containing those calls succeeds if all the calls succeed – otherwise it fails.

3.7. Statements

Statements are used to describe conditions, to perform output, to assign values to attributes, and to control the execution of the transformer via recursive subroutine calls. A statement is either a condition denoted by an expression, a call of a procedure, an assignment, one of the keywords REJECT or FAIL, or a target code statement. Every kind of statement may succeed or fail as described below.

```
Statement    = Expr | Ident ( [ Exprs ] [ => Patterns ] ) | Ident := Expr
             | REJECT | FAIL | [ Declarations ] TargetCode
Declarations = Parameters
```

Conditions are denoted by expressions and can be used to determine properties that can not be expressed with pattern matching alone. Patterns describe either shapes of a fixed size of a tree or the equality between two values. Properties of trees of unlimited size and relations like '<', '<=' etc. have to be checked with conditions. The expression has to be of type boolean or the call of a predicate. A condition succeeds if the expression evaluates to true – otherwise it fails.

For a procedure call the same rules as for a function call apply. It succeeds if the values of all output parameters are matched by the corresponding patterns – otherwise it fails.

An assignment statement evaluates its expression and stores this value at the entity denoted by the identifier on the left-hand side. The identifier can denote a global or a local variable, an input or an output parameter, as well as a label for an attribute or a subtree. An assignment statement succeeds if the expression succeeds – otherwise it fails.

The statement REJECT does nothing but fail. This way the execution of the current rule terminates and control is passed to the next rule. The statement FAIL causes the execution of the current subroutine to terminate. This statement is allowed in procedures and predicates, only. Depending on the kind of subroutine the following happens: A procedure terminates and a predicate returns false.

A target code statement which is enclosed in curly brackets '{' and '}' is executed as in the target language. It can be used to define labels by means of implementation language code or calls of external subroutines. In this case the names of the labels and their types have to be defined explicitly. This is done by declarations written in the syntax of a parameter list that precede the target code statement. A target code statement always succeeds.

Note, statements and expressions may cause side effects by changing e. g. global variables, local variables, the input tree, or by producing output. Those side effects are not undone when a rule fails.

Further features such as various possibilities concerning the use of hand-written target code, named patterns instead of positional ones, or the definition of the equality or matching operation between attributes by a macro mechanism are omitted for the sake of brevity.

4. Implementation

From a given specification, *puma* generates a program module in one of the target languages C or Modula-2 implementing the desired transformation. The subroutines in the sense of *puma* are mapped to subroutines in the target language. Procedures yield procedures, functions yield functions that return a value, and predicates yield boolean functions. These subroutines can be called from other modules using the usual subroutine call syntax of the target language provided they are exported: All arguments are separated by commas − the symbol '=>' as separator between input and output arguments is only required in calls processed by *puma*.

The types of the parameters are treated as follows: Predefined types or user defined types remain unchanged. Node types or sets of node types are replaced by the name of the corresponding tree type. This is a pointer to a union of record types. Input parameters are passed by value and output parameters are passed by reference.

The rules of a subroutine are treated like a comfortable case or switch statement. The code generated for pattern matching is relatively simple. A naive implementation would just use a sequence of if statements. This kind of code showed to be already rather efficient. *puma* optimizes the code with respect to the elimination of common tests for patterns and the clever use of switch statements. Furthermore, tail recursion can be turned into iteration. Labels are replaced by access paths to the associated values. The code for the construction of tree nodes is inserted in-line. It is therefore efficient because no procedure calls are necessary for the creation of tree nodes.

5. Related Research

In this section we compare our approach with several programming paradigms and similar tools. The intention is to provide further insight in the nature of our tool by looking at styles or work the reader might be familiar with.

5.1. Imperative Programming

Our approach can be regarded as an extension of imperative languages by a type constructor for trees. Operations on trees for the composition and decomposition are provided using a concise term notation. The storage management for trees is completely automatic. Intermediate variables for output parameters of subroutines are declared implicitly and checked for the single assignment property. Pattern matching represents a comfortable kind of case or switch statement tailored towards processing of trees. This imperative view is reflected best by the implementation of the generator *puma*. It can be seen as a preprocessor that provides attributed trees and pattern matching for imperative languages.

5.2. Logic Programming

From the stand point of logic programming languages such as Prolog [ClM84] our approach is relatively restricted and therefore one cannot classify *puma* as a logic programming language.

Puma has no backtracking, no logic variables, and nothing like assert and retract. The unification is unidirectional and restricted to one of the two terms being a ground term. Similar are the syntax and the presence of predicates. *Puma* retains the "imperative subset" of Prolog. It turns out that this subset can be implemented efficiently and it suffices to specify most of the transformation problems in compilers [Paa89].

Compared to Prolog the following features have been added: Besides predicates there are procedures, functions, and customary expressions. This allows for recursive functions and easy calls of *puma* subroutines from hand-written code. Attributes can be stored in the tree and accessed in read and write mode. The terms or trees are statically typed. For parameters the modes input and output are distinguished. Pattern matching is extended to cope with subtypes and attributes. The modification of input terms is possible. Easy escape to hand-written code and cooperation with other modules, either hand-written or generated, gives great flexibility and "imperative power". The transformer modules including the pattern matching are translated to direct code and therefore efficient.

5.3. Functional Programming

Compared to modern functional programming languages such as Hope [BMS80], ML [Mil85], or Miranda [Tur85] our approach can of course not claim to be functional. There is nothing like under or over supply of functions or higher-order functions. It just supports the restricted kind of functional programming that can be found in imperative languages having functions and recursion like e. g. C or Pascal. Even the function results are restricted to simple data types and pointers. However, the latter allow functions to return trees and graphs. With respect to the traditional functional language Lisp [MAE65] our approach might be worth to be compared. Whereas Lisp supports binary trees only, *puma* provides tree nodes with an arbitrary number of subtrees and attributes. Dynamic typing and binding have been replaced by their static counterparts. The pattern matching facility has been added.

5.4. Attribute Grammars

Our technique is related to attribute grammars [Knu68, Knu71] in several ways. Compared to pure attribute grammar processors such as [Gro89, JoP90, KHZ82] there are some restrictions. In pure attribute grammar systems the attribute computations are written in a functional notation. Knowing the dependencies among the attributes allows the automatic (implicit) determination of an evaluation order for the attributes (visit sequences). Furthermore it is possible to check an attribute grammar for completeness and non-circularity. Our approach is based on an explicit evaluation order using hand-written visit sequences. Three kinds of data can be regarded as attributes: The "real" attributes stored in the tree, the parameters of the subroutines, and global variables. This classification of the attributes is done by the user. Whereas pure attribute grammars allow only computations local to a grammar rule, the pattern matching facility makes computations on larger tree regions of fixed size feasible. Considering attributes of the kind parameter and non-nested patterns only, our approach is similar to affix grammars [Kos71, Kos77] or extended affix grammars [Wat74]. This holds from the syntactic as well as from the semantic point of view. A *puma* generated module can contribute to attribute evaluation in several ways: First, the evaluation can be carried out completely by a transformation module itself using one or more subroutines for traversing the tree. Second, *puma* subroutines can be called as external subroutines from an attribute evaluator to compute attributes − especially those depending on tree-valued arguments. The imperative or sequential style of *puma* allows simple control on side effects and easy production of arbitrary output.

5.5. Optran

The transformation tool Optran [LMW89] has been designed for optimizing transformations. It is based on an attribute grammar and modifies an input tree according to a given set of rules. A rule consists of a pattern, conditions, and action statements. The actions can replace the tree part matched by the pattern (where the unmatched subtrees might be reused) and compute new attribute values. Optran emphasizes the automatic reevaluation of attributes [LMO88] in order to arrive at attribute values consistent with the specified attribute grammar. As the goals of Optran are rather ambitious they pose several problems: In which order will the rules be applied and how to find the locations in the tree where a pattern matches? When will the reevaluation of attributes be carried out? The latter question is interesting because in the worst case it will be necessary to recompute many attributes in a large part of the tree and therefore consume a considerable amount of run time.

Our technique groups rules into an arbitrary number of subroutines. Every routine may perform pattern matching on an arbitrary number of trees supplied as arguments not just one. We explicitly control the execution order thus gaining an efficient and detailed way to describe where and when what should happen. *Puma* is not concerned with reevaluation of attributes. Besides modification of the input tree it also allows for mappings that produce arbitrary output.

5.6. Trafola

The functional language Trafola-H [HeS91] was designed to be a specification language for program transformations. Besides all the features one would expect from a modern functional language it has powerful constructs for pattern matching. For example pattern matching is extended to cope with tuples and sequences. This introduces nondeterminism which is handled using backtracking. The language is statically typed and offers type polymorphism. Pattern matching is implemented by a table-driven bottom up tree automata or by a backtracking algorithm. The functional constructs are usually implemented by interpretation of an internal abstract machine. Our approach does not aim to be functional but it retains the flexibility and efficiency of imperative programming. Whereas the Trafola type constructor for sequences allows the grammar for legal trees to be written in extended BNF, *puma* supports pure BNF, only. Both approaches allow so-called non linear patterns where a variable occurs more than once in a pattern. *Puma* has extended pattern matching to handle subtypes.

5.7. Codegenerator-Generators

Tools for the generation of code generators such as *Twig* [AGT89] or *Beg* [ESL89] often use tree pattern matching, too. In addition to a pattern, a condition, and an action a rule is associated with costs. Pattern matching usually performs a global match on the complete input tree in order to find a complete cover of the tree where the sum of all participating rules is of minimal cost. Algorithms based on dynamic programming have proved to yield satisfactory results. These tools are oriented towards code selection by transforming a tree-like intermediate representation into machine code. Therefore there is de facto only one subroutine with one tree-valued argument. *Beg* provides support for register allocation, has a fixed traversal scheme (post order), and generates directly coded code-generators. *Twig* allows for the modification of the input tree, supports arbitrary traversal strategies, and generates table-driven code-generators.

5.8. Txl

The tree transformation tool *txl* [CaC91, CoP90] processes concrete syntax. It comprises a parser and an unparser for input and output of trees and allows the generation of source-to-source translators. The tree transformation is described by a set of rules consisting of a pattern, a condition, and a replacement. Usually the user does not have to take care about the order or location of rule applications. Rules are applied as long as there is one that matches. Thereby the tree is modified.

There are means to describe which set of rules should be applied to which subtrees. Patterns are not written in a nested fashion but as strings containing nonterminals. The tool parses these strings in order to recognize the nesting structure. As *puma* is oriented towards abstract trees, the transformers are independent of parsing or unparsing. Besides tree modifications *puma* allows for tree mappings producing arbitrary output. Another difference is the explicit control of the execution order which allows an efficient implementation of the generated transformers.

6. Experiences

In a first real world application, *puma* was successfully used to generate a code-generator in a compiler for the robot control language IRL [IRL92]. The front-end of this compiler constructs an abstract syntax tree which is decorated with attributes during the semantic analysis phase. The code-generator maps this attributed tree to the standardized robot control code IRDATA [IRD91] which is comparable to assembly code. The strongly typed nature of IRDATA makes code-generation harder than one would expect. For instance it is impossible to implement record variables other than by turning every field into a separate variable. More complex types such as arrays with elements of a record type have to be transformed into records containing arrays. Nevertheless these nontrivial problems could be solved relatively easy with appropriate transformation rules. The code-generator uses a two phase scheme in order to deal with forward references. The first phase selects IRDATA instructions and stores them in memory. The second phase replaces symbolic labels by absolute ones, encodes the instructions, and writes the final code to an output file. The data structure for storing the instructions in memory is managed by a module generated by the tool *ast*. The development time was three months.

Table 1: Sizes of the Code-Generator Parts

	Specification [lines]	C Code [lines]	Binary Code [KB]
code selection (*puma*)	3032	8600	111
instruction storage (*ast*)	165	3430	36
output (hand-written)	-	700	7
total	3197	12730	154

The parts of the code-generator have the sizes listed in Table 1. The figures stem from measurements on a SPARC station ELC. The complete compiler runs at speed of 1000 lines per second. The run time is distributed among the phases as follows: Scanning and parsing takes 21 %, semantic analysis 10 %, code-generation 18 %, and output of the generated code 51 %. The huge share for the output comes from the voluminous nature of IRDATA. The output reaches four times the size of the source program.

7. Conclusions

We presented a tool for the transformation of attributed trees using pattern matching. It is based on the definition, composition, and decomposition of trees in combination with recursion and pattern matching for trees as well as for attributes. It supports the mapping of trees to arbitrary output as well as the modification of trees. Moreover it can be used to construct attribute evaluators. The easy escape to an imperative programming language assures flexibility and practical usability. The tool has been designed to provide an expressive technique that can be implemented efficiently.

The tool *puma* and its predecessor called *Gentle* [WGS89] have been used successfully in several real world projects. At our institution a front-end for a C compiler and a compiler for a functional language have been generated. An industrial company makes use of it for a language implementation project, too. All applications report their satisfaction with this approach and relative short development times. The interesting question is where does this success come from? The final paragraphs give the author's subjective opinion.

First, there are several aspects that support a high level programming style. The data type tree including operations for composition and decomposition replaces the handling of pointers, records, and dynamic storage allocation. Pattern matching offers a comfortable kind of case statement. Once it is understood and accepted then recursion seems to be easier to deal with then iteration. The implicit declaration of variables simplifies coding. The check for the single assignment property catches errors such as missing or multiple computations. A concise syntactic notation is probably more profitable than one would imagine. (Many of the the mentioned arguments have contributed to the success of Lisp and Prolog.) The approach supports an incremental development style: Initially, rules for the most general form are supplied and later rules for special cases are added in order to improve performance. Furthermore static typing discovers many errors during generation time.

Second, the approach is designed to be very flexible and open. Every combination of attribute processing from mere reading and matching of attributes to complete evaluation can be expressed. The method allows tree modifications as well as mappings that produce arbitrary output. The explicit description of execution order gives full control on side effects. The escape to imperative programming and the easy integration with external subroutines are essential loopholes. The tool as well as the generated code are efficient and can be used in production projects. It is probably the combination of all the mentioned properties and reasons that we regard this approach to be very promising.

References

[AGT89] A. V. Aho, M. Ganapathi and S. W. K. Tjiang, Code Generation Using Tree Matching and Dynamic Programming, *ACM Trans. Prog. Lang. and Systems 11*, 4 (Oct. 1989), 491-516.

[BMS80] R. Burstall, D. MacQueen and D. Sannella, HOPE: An Experimental Applicative Language, Report CSR-62-80, Computer Science Department, Edinburgh, 1980.

[CaC91] I. H. Carmichael and J. R. Cordy, *TXL - Tree Transformation Language, Syntax and Informal Semantics*, Dept. of Computing and Information Sciences, Queens's Univeristy, Kingston, Apr. 1991.

[ClM84] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer Verlag, Berlin, 1984.

[CoP90] J. R. Cordy and E. Promislow, Specification and Automatic Prototype Implementation of Polymorphic Objects in TURING using the TXL Dialect Processor, *Proc. IEEE 1990 International Conference on Computer Languages*, New Orleans, Mar. 1990, 145-154.

[ESL89] H. Emmelmann, F. W. Schröer and R. Landwehr, BEG - a Generator for Efficient Back Ends, *SIGPLAN Notices 24*, 7 (July 1989), 227-237.

[Gro89] J. Grosch, Ag - An Attribute Evaluator Generator, Compiler Generation Report No. 16, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1989.

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

[Gro91a] J. Grosch, Puma - A Generator for the Transformation of Attributed Trees, Compiler Generation Report No. 26, GMD Forschungsstelle an der Universität Karlsruhe, July 1991.

[Gro91b] J. Grosch, Tool Support for Data Structures, *Structured Programming 12*, (1991), 31-38.

[HeS91] R. Heckmann and G. Sander, Trafola-H Reference Manual, Prospectra Project Report, Universität des Saarlandes, Saarbrücken, 1991.

[IRD91] IRDATA, *Industrial Robot Data, DIN 66313*, Beuth-Verlag, Berlin, 1991.

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

[JoP90] M. Jourdan and D. Parigot, Application Development with the FNC-2 Attribute Grammar System, *LNCS 477*, (Oct. 1990), 11-25, Springer Verlag.

[KHZ82] U. Kastens, B. Hutt and E. Zimmermann, *GAG: A Practical Compiler Generator*, Springer Verlag, Heidelberg, 1982.

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

[Knu71] D. E. Knuth, Semantics of Context-free Languages: Correction, *Mathematical Systems Theory 5*, (Mar. 1971), 95-96.

[Kos71] C. H. A. Koster, Affix Grammars, in *ALGOL 68 Implementation*, J. E. L. Peck (ed.), North Holland, Amsterdam, 1971, 95-109.

[Kos77] C. H. A. Koster, CDL: A Compiler Implementation Language, *LNCS 47*, (1977), 341-351, Springer Verlag.

[LMO88] P. Lipps, U. Möncke, M. Olk and R. Wilhelm, Attribute (Re)evaluation in OPTRAN, *Acta Inf. 26*, (1988), 213-239.

[LMW89] P. Lipps, U. Möncke and R. Wilhelm, OPTRAN - A Language/System for the Specification of Program Transformations, System Overview and Experiences, *LNCS 371*, (1989), 52-65, Springer Verlag.

[MAE65] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. R. Hart and M. I. Levin, *Lisp 1.5 Programmer's Manual*, MIT Press, Cambridge, MA, 1965.

[Mil85] R. Milner, The Standard ML Core Language, *Polymorphism 2*, 2 (1985), .

[NAJ76] K. V. Nori, U. Ammann, K. Jensen, H. H. Nägeli and C. Jacobi, The Pascal-P Compiler: Implementation Notes, Bericht 10, Eidgenössische Technische Hochschule, Zürich, July 1976.

[Paa89] J. Paakki, A Prolog-Based Compiler Writing Tool, in *Proceedings of the Workshop on Compiler Compiler and High Speed Compilation*, D. Hammer (ed.), Berlin, GDR, 1989, 107-117.

[Tur85] D. A. Turner, Miranda: A Nonstrict Functional Language with Polymorphic Types, *LNCS 201*, (1985), 1-16, Springer Verlag.

[Vol91] J. Vollmer, The Compiler Construction System GENTLE, GMD-Arbeitspapier Nr. 508, GMD Forschungsstelle an der Universität Karlsruhe, Feb. 1991.

[WGS89] W. M. Waite, J. Grosch and F. W. Schröer, Three Compiler Specifications, GMD-Studie Nr. 166, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1989.

[Wat74] D. A. Watt, Analysis Oriented Two Level Grammars, Ph. D. thesis, University of Glasgow, Glasgow, 1974.

Contents

	Abstract	1
	Keywords	1
1.	Introduction	1
2.	Tree Transformation by Pattern Matching	3
3.	Specification Language	5
3.1.	Tree Structure	5
3.2.	Subroutines	6
3.3.	Types	6
3.4.	Rules	6
3.5.	Patterns	7
3.6.	Expressions	7
3.7.	Statements	8
4.	Implementation	9
5.	Related Research	9
5.1.	Imperative Programming	9
5.2.	Logic Programming	9
5.3.	Functional Programming	10
5.4.	Attribute Grammars	10
5.5.	Optran	11
5.6.	Trafola	11
5.7.	Codegenerator-Generators	11
5.8.	Txl	11
6.	Experiences	12
7.	Conclusions	12
	References	13