A Metalanguage To Express Human Guidance for Program Transformation

Jin Lingzi and Zhu Hong

Computer Science Department
Nanjing University, Nanjing, P.R.China

Abstract The researches on transformational programming and transformation system have an important impact on the improvements of both software quality and productivity. Among various approaches, metalanguage one has advantages of automating the transformation process and widening application area at the same time. It is achieved by systematic management of design decisions according to the knowledge and intuition of human being expressed in metaprograms. This paper reports a metalanguage TrapML and an abstract transformation system NDTPS, and some experiments with them.

1. Introduction

Transformational programming has attracted growing attention for its advantages of high reliability, reusability and capability of mechanical support. However, it is very difficult and complicated to develop software by applying rules step by step. The transformation systems developed so far usually depend on some special methods and techniques, or on the fixed transformations and strategies. They are still far away from the practical uses.

The problem perhaps comes from the difficulties in the organization and use of knowledge about programming and application domains. It is widely accepted that such knowledge is of particular importance in programming. In each step of the transformation process the system should decide which part of the expression should be changed, which rule should be used, and so on. However, there is a gap between the knowledge and making decisions in transformation. Fixed rule set and transformation strategy seem impossible to cope with complicated situations in the development of practical software. So the key point is how to express knowledge and intuition of human being to guide transformation process. The metaprogramming is perhaps among the most promising solutions. Because a metaprogram provides knowledge and controls the transformation process, various decisions can be systematically organized.

In this paper, a metalanguage TrapML is proposed which has the facilities to describe the syntax of object languages, transformation rules and strategies. Metaprograms are used to equip the system forming a specific transformation system. Some experiments with metaprogramming have been carried out such as the construction of a transformation system for optimization of Backus’ FP programs, the description of synthesis process of divide-and-conquer algorithms from pre/postconditions, an interpreter of equational logic, and a synthesis system which derives linear recursive FP programs from input/output examples.

2. The Metalanguage--TrapML

The structure of TrapML is the combination of three facilities to describe the syntax of object languages, transformation rules and strategies. A metaprogram consists of a series of syntax modules, rule set modules and strategy modules. A syntax module independently defines the syntax structure of an object (sub)language, which is referred to by rule set modules to form transformation rules. Transformation strategies are described in strategy modules. If they are only applied to some special languages and/or rules, the related syntax and/or rule modules are involved, or they are general and do not refer to other modules which means that they depend on neither languages nor rules.

The following is a brief introduction to the structures and functions of these facilities. The details of the syntax and semantics of TrapML can be found in [6].

2.1. Syntax description modules.

Syntax description modules specify the syntactical structure of object languages. Since programs are constructed from basic components by program forming operators (PFOs), a syntax module specifies the structure of a language by indicating the set of PFOs. Its basic components are 0-ary PFOs. To improve the reliability of metaprograms, users can assign sorts to expressions of the language. The overall structure of the syntax module is:

< syntax_module> ::= 
SYNTAX < name_of_language> ;
SORT < list_of_sort_name> ;
PFOS < list_of_PFO_dec>
END;
< list_of_sort_name> ::= 
< sort_name> | < sort_name> .
where sort names are identifiers.

There are five forms of PFOs:

- **prefix form**: PREFIX < operator > (S₁,S₂...,Sₙ) : Sₚ
- **postfix form**: POSTFIX # < operator > : S₁ -> Sₚ
- **infix form**: INFIX
- **infixa form**: INFIXA
- **distributive-fix form**: DISTFIX

**where Di, i=0…n, and Sᵢ are sorts.** Of particular,

**which specifies a 0-ary operator.**

Infix operators with associativity can be declared by associating infix declarations in the infix form before the PFO with a list of expressions of sort S. For example, the functional construction of Backus' FP system can be described as follows:

```plaintext
FORALL < list_of_variable_dec > THAT
  < scheme > < direction > < scheme >
  [ WHERE < condition > ]
```

where transformation direction may be ‘>’ which means transforming from left to right, or ‘=’ denoting both -> and in the opposite. Variables used in the schemes and conditions must be declared. They are local to the rule. There are two kinds of variable declarations, the declarations of single variables and variable families.

The declaration of single variables is in the form of

- \( v_1, v_2, \ldots, v_n : S \)

where \( v_i \) 's (i=1,2,..,n) are identifiers and S is a sort name. It declares n variables \( v_1, v_2, \ldots, v_n \) of sort S.

The declaration of a variable family is in the form of

- \( \forall \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n : \mathbf{v}_1(i) \leftarrow u_1(I), \mathbf{v}_2(i) \leftarrow u_2(I), \ldots, \mathbf{v}_n(i) \leftarrow u_n(I) : S \)

which declares k families of variables \( \{v_i[I] \mid 1 \leq i \leq n \} \). or (\( \forall \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n : \mathbf{v}_1(i) \leftarrow u_1(I), \mathbf{v}_2(i) \leftarrow u_2(I), \ldots, \mathbf{v}_n(i) \leftarrow u_n(I) : (k \times 1) \)

where \( v_i[I] \) and \( u_i \) are lower and upper bounds respectively. A lower bound must be a constant of number, while an upper bound may be a constant or a variable of number.

Schemes are well-formed expressions with occurrences of variables. Moreover, quantified schemes are designed to describe structures of programs formed by PFOs containing repetitive components. A quantified scheme is in the form of

- \( \forall i = 1..u \) < scheme >

where i is a bounded variable ranging over the set \( [1, l+1, \ldots, u] \). It can be used in the scheme as an expression of sort number, or as an index to access an element of variable family. Quantified schemes are only used as a component of a scheme which is formed by a PFO containing repetitive components. The lower bound l must be a constant of number, while the upper bound may be a constant or a variable of number. Whenever the upper bound is fixed, we can substitute i by constants \( l+1, \ldots, u \) into the scheme, and obtain schemes \( M(l), M(l+1), \ldots, M(u) \). Then the meaning of the quantified scheme is that the program formed by the PFO has a sequence of subexpressions \( E_i, E_{i+1}, \ldots, E_{i+n} (n = u-l) \) such that for every i in \( [0, l, \ldots, n]\), the subexpression \( E_i \) is matchable with the scheme \( M(l+i) \). It is why we call it quantified scheme. For example, the following scheme

- \( \forall i = 1..n \) f(i) o l(i) \)

where n is a variable of sort number, each f(i), i = 1,2,..,n, is an element of a family of variables, l(i) is a selection function of FP system, is matchable with the following FP programs

- \( F_1 \circ l(1), F_2 \circ l(2), \ldots, F_n \circ l(n) \)

for all programs \( F_1,F_2,..,F_n \) in FP system and natural number n.

Given a scheme M and a program segment P, if there is a substitution \( \sigma \) such that an ordinary scheme \( M' \) can be obtained by replacing every variable x of sort number
by $\sigma(x)$ to determine the number of variables in variable
families and upper bounds of quantified schemes. If
$\sigma(M) = P$, we say that $P$ is matchable with $M$.

The introduction of quantified schemes considerably
enhances the expressivity of patterns. It can be used to
describe various structures of lists. For example,

(1) the structure that all elements are in the same form,

(2) the structure that there is one element in a given form,

(3) the mixture or nest of the above two, say,

Other advantages of quantified schemes are that they are
directly perceived through the senses, and pattern matching
can be done effectively.

In transforming a program segment, if there is a
substitution $\sigma$ which makes the left-hand side scheme of a
rule matchable with it, and the condition in where-clause of
the rule is satisfied under $\sigma$, the rule is applicable. In this
case, if the application of the rule is asked, then the instance
of the right-hand side scheme takes the place of the
segment.

The condition is constructed from simple conditions
by logical connectives 'and', 'or' and 'not'.

(1) Selection of rule sets, such as

(2) Selection and application of rules.

In TrapML, the only manipulation of programs is
the application of rules. The correctness of metaprog-
grams can be localized in rule set modules. Therefore, relia-
bility of metaprogms can be obtained. Since it is the
basic facility to describe human guidance, a set of applica-
tion primitives are provided for various usage of rules.

apply \[ < - I \rightarrow ] < rule_name> \[< withclause> ] \rightarrow

(3) Change of the transformation focus.

A program is regarded as a tree. The transformation
focus is then the root of the subtree of the program
segment being transformed. Thus, the change of the focus
is just the movement in the tree. TrapML provides
several such primitives, for example,

(4) Version control.

In order to organize search and backtracking effectively,
intermediate versions of programs can be controlled
by strategies, such as

save \[< name> ] \rightarrow save the program being
transformed into version library with the name. When the
option name is absent, a version number is assigned to by
the system.

back \[< name> ] \rightarrow restore the version with the name
from version library as the new program to be
transformed. If the option name is absent, the last version saved is restored.

The control structures include the following. Conditional: IF < predicate> THEN < strategy> ELSE < strategy> END

It selects either of the strategies according to the value of the predicate.

Predicates are constructed from primitive predicates by logical connectives 'and', 'or' and 'not'. Primitives include ones to test the structure and context of a program segment, such as, leftmost, rightmost, have_father, have_son, name(< operator>), sort(< sort_name>), etc; and ones to test transformation rules, such as positive_direction, next_match, and so on. FindAR is also a predicate whose value is true if there is an applicable rule otherwise false.

Scheme directed branching:
CASE < scheme_name> : < strategy>
{ [< scheme_name> : < strategy> ]}
[ OTHERWISE < strategy> ] END,

The schemes whose names are in the list are matched with the current focus sequentially until the one matching the segment is found, then the corresponding strategy is selected to guide further transformation. If there are no matchable schemes in the list, the strategy in otherwise_clause is used if it exists.

Sequential Composition: < strategy> ; < strategy>
The strategy S1; S2 means that when the strategy S1 is finished, the S2 is the subsequent transformation strategy to be used.

For all branches: ATOALL < strategy> END
It uses the strategy to transform every branch of the current focus.

Construction: [ < strategy> [ , < strategy> ] ]
When a strategy in the form of [S1,S2,...,Sn] is used, then S1 is used to guide the transformation of the i'th branch of the segment if there is such a branch. When there are more than n branches in the segment, branches with number greater than n are left unchanged.

Repetition: WHILE < predicate> DO < strategy> END
The strategy is used repeatedly whenever the predicate is true.

Algorithm family generation:
FAMILY < version_name> : < strategy>
{ [< version_name> : < strategy> ]}
END.
Transform the program segment according to the strategies respectively and generate a family of versions with corresponding names.

Strategies can be declared by definitions.
DEF < def_name> [ ( < list_of_parameter_dec> ) ]
= < strategy>
The definition of strategies may be parameterized. There are four types of parameters, the rule set, rule, scheme, and strategy.

< list_of_parameter_dec> ::= < parameter_dec> ; < parameter_dec> }
< parameter_dec> ::= rule | ruleset | scheme | strategy.

Formal parameters are used in the definition body as rule name, rule set name, scheme name, and strategy respectively. The call of definition is in the form of
< definition_name> { < list_of_actual_parameters> }

In the call of the definition, actual parameters are given which then take the place of formal parameters in the body. They must be compatible with formal parameters, i.e. rule name for the formal parameter of type rule, rule set name for the parameter of ruleset, and so on.

The strategy definition and call provide a facility of strategy abstraction which may be (mutually) recursive and parameterized.

The following are some simple but powerful recursive strategies.

DEF exhaustive =
WHILE find_AR DO application END;
ATOALL exhaustive END;

DEF outmost_leftmost =
IF find_AR
THEN application;
root;
outmost_leftmost
ELSE ATOALL outmost_leftmost END
END

3. NDTPS -- An Abstract Transformational Programming System

NDTPS is a software system to support the transformational development of software. It is an abstract transformation system in the sense that it is independent of concrete and abstract syntax of objects, transformation rules, and control strategies. Its structure is illustrated in Fig.1.

The library of metaprograms stores the syntactical structures of objects, transformation rules and strategies. Version library holds various versions generated in transformation process. The transformation engine looks for an applicable rule, and applies it when necessary. The translator is made up of four parts, the translator of the set of PFOs, of programs and their schemes, of transformation rules and of strategies. A retranslator will convert the internal representation of a program into its external context. Users select commands from menus in the man machine interface, and the contents of libraries are displayed in windows. The main functions of control unit is that it

(1) controls the engine and managers;
(2) exchanges information and commands with the
man-machine interface; (3) records transformation process.

Fig. 2. The Structure of NDTPS.

Metaprograms are used to equip the system. The syntax modules are translated into the internal form and put into PFO base. The rule set modules are then translated into internal form according to the contents in PFO base. Strategy modules are translated into series of basic commands provided by the control unit and then stored in strategy library. They can be called as high level commands of the system and executed automatically by the control unit. Therefore, the system has two states, the metaprogram guided state and the interactive state. Exchange between the two states is flexible. Users can enter the metaprogram state at any time of interaction by calling one of strategies in the library, and be back if the primitive halt or exit is met in the execution of a metaprogram.

More details about the system can be found in [6,7].

Some Experiments with metaprogramming in TrapML have been made for example, (1) The construction of an FP transformation system [7].

A transformation system to optimize Backus' FP programs has been constructed on the NDTPS by metaprogramming. It uses the algebra of programs to specialize a general program on some specific data.

(2) The expression of divide and conquer programming paradigm [6].

The knowledge of divide-and-conquer programming is expressed in TrapML. The execution of the metaprogram has derived some programs from pre and post-condition specifications, such as the one to compute the maximal element of a list of numbers.

(3) The interpretation of the equational logic as a programming language [6].

An interpreter of equational logic can be built easily in TrapML by writing equations in rule set modules. The outermost leftmost computation strategy can be defined in strategy module.

(4) The implementation of an FP synthesis system.

The system synthesizes linear recursive programs of Backus' FP system from input/output examples. It implements a knowledge based synthesis methodology [8] based on a theory of orthogonal expansion of functional programs [9,10]. The metaprogram contains a dozen of modules and about one hundred rules. Some meaningful programs such as inner product of vectors and matrix multiplication are synthesized.

4. Conclusion

Although only a little experiments with metaprogramming have been reported [11,14], this approach has been considered to be one of the most promising ones to transformational programming [4].

From our experiments, we are convinced that

(1) The flexibility of the metalanguage approach greatly helps the effective organization of transformation process, and the expression of knowledge in an abstract and high level. It provides a vehicle to express human guidance to transformation process.

(2) High level metaprograms can make the system concise, easy to understand, modify and verify. A carefully designed metaprogram also results in the high efficiency since less search and backtracking are needed.

(3) High reusability of metaprograms can be achieved. Some vague and general strategies are reusable if they are less dependent on specific syntax and rules. The separation of syntax, rule set and strategy in different modules offers the possibility to use same syntax and rule sets in several strategies. For example, the syntax module FP_system and rule set module FP_program_algebra are used in the FP transformation system and the FP synthesis system.

REFERENCES


Program Transformation Approaches and Techniques, in [1], 165-195.


