PROSPECTRA
an ESPRIT Project

Notes on TRAFOLA, I

A Proposal for the Syntactic Part of the PROSPECTRA Transformation Language

Reinhold Heckmann

Universität des Saarlandes
6600 Saarbrücken
Bundesrepublik Deutschland

PROSPECTRA Study Note S.1.6-SN-6.0

1987-02-17

Distribution: confidential

ABSTRACT

After a short introduction into the problems of the current OPTRAN system, we shall propose more powerful patterns with type constraints and wild cards to match tuples of arbitrary length and tree fragments (contexts) of arbitrary height. Then a syntax for a functional transformation language is proposed and finally, a complex transformation (removal of function calls) is described.

This study note was developed by the author in many discussions with P. Lipps, D. Maurer, U. Möncke, F. Müller, H.G. Oberhauser, S. Pistorius, B. Weisgerber, and R. Wilhelm.

Reinhold Heckmann
Universität des Saarlandes

in the Project

PROgram development by
SPECification and
TRAnsformation

sponsored by the

Commission of the European Communities

under the

European
Strategic
Programme for
Research and development in
Information
Technology

Project Ref. No. 390

1. Disadvantages of the current OPTRAN system and goals for its reform

1.1. Semantic actions

Semantic actions and predicates must be written in Pascal, an imperative language without functions of higher order, without abstract data types or modularization, without generic types parametrized by type variables, and without overloading.

Pascal does not provide means for garbage collection, the user him/herself is responsible to maintain the storage for attribute values. Using pointers and assignments allows for messy side effects making semantic functions mysterious for the system; OPTRAN cannot itself infer efficient realizations of semantic actions. On the other side, the use of side effects is promoted by the requirement of writing efficient semantic actions. The final efficiency and even correctness of the generated program is thus left at the user's own responsibility.

A solution would be to provide a powerful functional language leading to transparent semantic actions that may be efficiently performed by OPTRAN. This language should combine facilities of a general purpose functional language with powerful features to express transformations of trees and computation of attribute values.

After having discussed some general aspects of the problem, we shall consider possible features of a future transformation language in the remainder of this note. It has turned out that the power of patterns will be one central point of this language, the other main question will be how to combine transformations to more complex ones.

1.2. Transformation rules

Actual transformation rules consist of an input pattern matching some subtree t of an enclosing tree t_o and binding some parameter names to some subtrees of t, of a predicate p controlling the applicability of the rule, of an output pattern describing the shape of a new subtree t', and of rules for computing some attributes of t'. The semantics of the rule applied to some tree t_o is - roughly spoken - looking for a subtree t suitable for matching the input pattern, then testing the predicate p on it. If it yields false, nothing happens, otherwise the subtree t is replaced by t' and the syntax tree t_o is reattributed.

The semantics of the application of the whole set of rules to a tree object could be seen in a non-deterministic way. Each rule looks for suitable subtrees and replaces them independently from the other rules. Non-determinism can be eliminated by introducing global strategies such as "top down, left to right" to define the order of considering subtrees, and the convention to try to apply atomic rules at each individual subtree in their textual order.

Thus there is no developed hierarchy on transformation rules, there are only two levels: atomic rules and the set of all rules. Mechanisms should be provided to combine atomic rules to more complex ones, and there should be more flexible and powerful means of controlling the order and effect of transformation rules. In addition, the atomic rules themselves could be made more powerful by raising the expressive power of input and output patterns.

A notation for "apply rule r at given subtree t" should be developed for the interactive version of OPTRAN to make it still more suitable for supporting the PROSPECTRA methodology.

1.3. Uniformity of different mechanisms

In the current OPTRAN system, transformations of syntax trees are described by a set of atomic rules of a special form, and semantic actions (transformations of attribute values) and predicates are described by Pascal procedures. It would be advantageous to provide uniform tools for both tasks. Then it would be possible to consider attribute values as trees that are again attributed, and so on. Thus, a possibly infinite tower of levels of attribution would result.

Another point is to consider transformations (of trees and of attribute values) as object of further transformations by meta-rules. In order to avoid self modifying programs, two strictly different system modes should be distinguished: a mode where a set of fixed transformations resp. transformation scripts (programs in a transformation language) is applied, and a mode where rules/scripts may be changed or new rules/scripts may be added incrementally.

The current OPTRAN system consists of a tree transformer and a generator producing the transformer from a so-called t-unit. The first mode mentioned above corresponds to a run of a generated tree transformer, and the second mode currently consists of text editing on a t-unit by some text editor and then running the generator again.

Means should be provided to do the development and incremental update of transformations and scripts inside OPTRAN by a special meta-transformer incorporating the meta-rules.

Since transformations and their semantics should be sufficiently well known, we hope that there will be no requirements for meta-meta-rules by the PROSPECTRA methodology, thus the meta-transformer will be fixed once for all time.

In the following chapters, we shall consider features to describe sample transformations without attribution. The nature of attributes will be considered in a later study note. We think that at first a suitable base (notion of trees and transformations) must be looked for before the two hierarchies mentioned above can be explored.

2. Tree grammars

2.1. Basic notation

In current OPTRAN, a tree grammar is a set of non-terminals, a set of operators with given arity, and a set of productions

X ::= Y
X ::= < op >
X ::= < op, X1, ..., Xn >

This notation is not very suitable for programming transformations in a functional style. Thus we propose to replace it by a signature notation:

non-terminal N	sort N
terminal T	terminal sort T
operator op	constructor op
X ::= Y	Y $\subseteq$ X
X ::= < op >	op: X
X ::= < op, X1, ..., Xn >	op: X1 X $\cdots$ X Xn $\rightarrow$ X

A terminal sort is a collection of atomic objects e.g. Num = {0, 1, 2, ...}. We call the union of all syntactic sorts Term. Term is the set of all terms of constructors, it corresponds to the set of all possible subtrees in the traditional notation.

2.2. An example grammar

Now we introduce an example grammar for use in later discussion of transformations.

Grammar 1

Syntactic sorts: Var, Exp, Stm
Terminal sorts: Num = {0, 1, 2, ...}, Id = {i1, i2, i3, ...}
Constructors: see below
Relations between them:
Id $\subseteq$ Var
Var $\subseteq$ Exp
Num $\subseteq$ Exp
false, true: Exp
sign, not: Exp $\rightarrow$ Exp
add, sub, mul, div, and, or, le, eq: Exp X Exp $\rightarrow$ Exp
noop: Stm
assign: Var X Exp $\rightarrow$ Stm
if: Exp X Stm X Stm $\rightarrow$ Stm

while:	Exp X Stm $\rightarrow$ Stm
sep:	Stm X Stm $\rightarrow$ Stm

Alternatively, these rules could be written more traditionally as

Var ::= Id
Exp ::= Var | Num | false | true | sign Exp | not Exp
  | add (Exp, Exp) | etc.
Stm ::= noop | assign (Var, Exp) | etc.

A useful extension is to allow for building simple constructor sorts from a finite set of constructors, e.g.

Binop = {add, sub, ...}
Binop: Exp X Exp $\rightarrow$ Exp

A typical element of Term is

sep (assign (i1, sign i2),
  while (le (i1, i3), assign (i1, add (i1, 1))))

Trees might be more readable if prefix notation is replaced by infix notation with delimiters (for unambiguity):

add (Exp, Exp)	(Exp + Exp)
sem (Stm, Stm)	(Stm ; Stm)
while (Exp, Stm)	while Exp do Stm od
etc.	

The example term above would then be written

((i1 := sign i2) ; while (i1 <= i3) do (i1 := (i1 + 1)) od)

Infix notation is likely to be useful for the user interface, in which case it must be computed by a deparser.

In the following we shall always use prefix notation in our examples.

2.3. Generalization of tree grammars

The form of productions need not be restricted to the patterns mentioned above. One further form seems to be particularly useful:

op: X1* $\rightarrow$ X2   resp. X2 ::= op X1*

For syntactic sort S, sort S* is defined by

$$S * = \{ () \} \cup S \cup S X S \cup S X S X S \cup \cdots$$

thus $S \subseteq S *$, $S X S \subseteq S *$, etc. hold, and there is no difference between a term and a one element list, a pair and a two elements list etc.

In concrete terms, elements of S* are written

( ) (s1) (s1, s2) (s1, s2, s3) etc. where s1, s2, s3 $\in$ S.

The special element "()" is called empty tuple; it is not typed and thus belongs to all starred sorts.

In a production "op: X1* $\rightarrow$ X2", "op" is a so-called list operator with arbitrary arity, requiring operands to be all of the same type. Directly left or right recursive systems of productions may be replaced by such iterations.

The presence of list operators allows for more compact, readable, and powerful transformation rules, so we think the feature of list operators is to be included in future OPTRAN.

To have examples with list operators, we introduce example grammar 2:

Grammar 2:

The only difference to grammar 1 is that the very last production	
sep: Stm X Stm $\rightarrow$ Stm	resp. Stm ::= sep (Stm, Stm)
is replaced by	
statlist: Stm* $\rightarrow$ Stm	resp. Stm ::= statlist Stm*

Some further extensions of tree grammars should be discussed but are not likely to be useful:

1) Rules c: R $\rightarrow$ S where c is a constructor, S a sort and R an (arbitrary or restricted) regular expression of sorts built up by $\cup$, X, *.
2) Rules R $\subseteq$ S where S is a sort and R a (restricted) regular expression.

2.4. Operators

The restriction to productions of the shapes

(1)	X ::= Y	resp. Y $\subseteq$ X
(2)	X ::= op	resp. op: X
(3)	X ::= op (X1, ..., Xn)	resp. op: X1 X $\cdots$ X Xn $\rightarrow$ X
(4)	X ::= op Y*	resp. op: Y * $\rightarrow$ X

where different productions must contain different operators allows for introducing a mapping "arity" from the set of all operators to the set $\mathbf{N} \cup \{*\}$. Operators occurring in form (2) have arity 0, in form (3) n, and in form (4) *.

In terms, we shall notationally distinguish between nullary operators and list operators without children, but terms of arity n > 0 and actual list terms having n subterms look equal.

0 children:	true	statlist ()
1 child:	not true	statlist s1
or	not (true)	statlist (s1)
2 children:	add (num, num)	statlist (s1, s2)

3. Transformations

3.1. Some sample transformations

The following transformations on terms of grammar1 and grammar2 seem to be useful:

(if-true)	if (true, T, F)	into	T
(if-false)	if (false, T, F)	into	F
(while)	while (false, S)	into	noop

Those rules are so simple that they can also be formulated in current OPTRAN. Now let us consider some more difficult transformations in terms of grammar 1.

Assume that the standard form of statement lists associates to the left such that "S1; S2; S3" is parsed into "sep (sep (S1, S2), S3)". This standard form may be violated by (if-true) and (if-false) if the result T resp. F is a "sep" term and the original "if" term is also subterm of a "sep" term. In addition, we want to get rid of the "noop" produced by (while) whenever this is possible.

(sep)	sep (S1, sep (S2, S3))	into	sep (sep (S1, S2), S3)
(noop1)	sep (S, noop)	into	S
(noop2)	sep (noop, S)	into	S

These three rules may also be formulated in current OPTRAN, but it is not possible to describe

- the semantic similarity of (if-true) and (if-false) - both serve to simplify degenerated "if" terms - , and of (noop1) and (noop2) - both delete a superfluous "noop" statement.
- the fact that a successful application of (if-true) or (if-false) at one position implies that (sep) is likely to

be applicable at a related position. The same relation holds for (while) and (noop1) and (noop2).

- the fact that (sep) once it is applicable should be applied repeatedly until the left recursive standard form is reached.
- their role as normalization rules like the "laws" in Miranda [2]. They should be silently applied by OPTRAN whenever and wherever it is possible.

The sequencing of rules is only implicit in current OPTRAN and cannot be explicitly expressed. This problem will be delayed until chapter 5 since we first want to try to make the atomic rules as powerful as possible.

3.2. Semantics of transformations

We shall use the following syntax for transformations:

Pattern => Expression

where "=>" replaces the former "into".

"Pattern" and "Expression" are syntactic sorts of the transformation language to be developed (don't confuse "Expression" with "Exp", a syntactic sort of the example grammars 1 and 2). What is the semantics of the rule "p => e" applied to some value v?

For the moment, the only values are terms, but the set of values will be extended by further generalizations. If p does not match v, the rule fails, otherwise the parameter names occurring in p are bound to values (subterms of v). Thus p applied to v returns an environment r. Then the expression e is evaluated in this environment to a new value v'. The transformation "p => e" thus describes a partial mapping of values.

Later we shall consider non-deterministic patterns that may match in different ways thus returning a set s of environments when applied to a value v. The failure case fits well in this view, the pattern then returns an empty set of environments. The expression e is evaluated in each environment contained in s and yields thus a set of values.

Environments:	$\mathrm{Env} = \mathrm{Name} \rightarrow (\mathrm{Value} \cup \{\mathrm{unbound}\})$
Semantics of patterns:	$\mathbf{P}: \mathrm{Pattern} \rightarrow \mathrm{Value} \rightarrow 2^{\mathrm{Env}}$
of expressions:	$\mathbf{E}: \mathrm{Expression} \rightarrow \mathrm{Env} \rightarrow \mathrm{Value}$
resp.	$\mathbf{E}: \mathrm{Expression} \rightarrow 2^{\mathrm{Env}} \rightarrow 2^{\mathrm{Value}}$

We shall adopt the non-deterministic view in the sequel since it is easier to describe. Determinism may be achieved by imposing decision strategies to designate one kind of matching if more than one is possible. But we think that these strategies are independent from the basic semantics of patterns and may be developed later.

4. Patterns

In the following presentation, we shall obey some conventions:
- lower case letters denote values or patterns
- upper case letters are parameter names to be bound by the pattern
- lower case words are operators such as "if", "noop"
- capitalized words are syntactic sorts such as "Stm", "Exp"

4.1. Simple patterns

Simple patterns are just terms where some subterms are replaced by parameter names. The pattern matches terms of this shape and binds the names to the corresponding subterms.

"if (true, T, E)" matches "if" terms and binds T and E to "then" resp. "else" part.

"T" matches any term and binds T to it.

Simple patterns are the only kind of patterns allowed in current OPTRAN. They are always deterministic.

4.2. Typed patterns

In current OPTRAN, there is no way to express that we want to match only terms of a specific syntactic sort. To overcome this, we propose typing of parameter names by syntactic sorts.

"S: Stm" matches any statement and binds S to it.

"if (true, T: Stm, E: Stm)" is equivalent to "if (true, T, E)" due to the structure of grammar 1.

If we introduce a special symbol '_' to denote the sort Term, i.e. the union of all syntactic sorts, then we are able to require without loss of generality that all parameter names be typed in patterns.

"if (true, T: _, E: _)" instead of "if (true, T, E)".

The abbreviation of "X: _" to "X" could then be introduced as alternative paraphrasing.

Types need not be simple sorts or '_':

"I: if (true, Stm, Stm)" and "I: if (true, _, _)" both match "if" terms with condition "true" and bind I to the whole "if" term.

"I: if (true, T: Stm, Stm)" shows that there is no difference between types and patterns. It matches "if true" terms and binds I to the whole term and T to the "then" part. If we write "I: if (true, T: _, _)" or "I: if (true, T, _)" by alternative paraphrasing we see that our underscore '_' is much like the one in Hope or Prolog - a don't care pattern or wild card for terms.

4.3. Tuple patterns and values

Assume we want to write a rule to delete superfluous "noop" statements in statement lists in terms of grammar 2. Then we need a rule

statlist (L1: Stm*, noop, L2: Stm*) => statlist (L1, L2)

In this rule, we have used patterns with iterators (here "Stm*") that match tuples of terms with arbitrary length. Thus L1 and L2 are bound to tuples, and comma means concatenation of tuples in patterns and expressions, i.e.

((a, b, c), (d, e, f), (), g) = (a, b, c, d, e, f, g)

Values are now tuples of terms, thus names are bound to tuples by patterns, and patterns may be applied to tuples.

Patterns and values are concatenated by commas. In a value like "if (true, noop, noop)" the subvalues "true" and "noop" form 1-tuples that are concatenated to a triple "(true, noop, noop)". This triple is mapped into a term by the "if" operator.

Examples for patterns:
"statlist (L1: _*, statlist (L2: _*), L3: _*)" matches statement lists of arbitrary length containing an element that is again a statement list of arbitrary length.
"statlist (L1: _*, statlist (L2: _), L3: _*)" matches statement lists containing an element that is a statement list of length one consisting only of the term "L2".
"statlist (L1: _, noop, L2: _)" matches only lists with exactly 3 elements: "L1", "noop", and "L2".
"statlist (L1: _, noop, L2: _*)" matches lists whose length is at least 2 and whose second element is "noop".
"statlist (noop*)" matches lists of arbitrary length where all elements are "noop" statements.
"if (id, R: _*)" matches all "if" terms whose condition is "id". R will always be bound to a pair since "if" has arity 3.

Syntax of patterns:

Pattern ::= _ | () | Sort | Name : Pattern | c0 | c Pattern
| Pattern , Pattern | Pattern Iterator

where c0 is a nullary operator and c an operator of arity > 0. Parentheses ( ) are to be used to solve ambiguities. The empty parentheses "()" match only empty tuples.

Iterator ::= { i .. j }   where $i, j \in \mathbf{N}_0 \cup \{\infty\}$.

Proposals for alternative paraphrasing of iterators:

{i .. }	{i .. ∞}
{ .. j}	{0 .. j}
{ i }	{i .. i}
*	{0 .. ∞}
+	{1 .. ∞}
?	{0 .. 1}

Comma operator and iteration are a potential source of non-determinism:
statlist (L1: Stm*, noop, L2: Stm*)
applied to the value
statlist (a1, noop, a2, a3, noop)
where the subvalues ai are statements, but not "noop" statements, yields a set of two environments:
[L1 → a1; L2 → (a2, a3, noop)] and
[L1 → (a1, noop, a2, a3); L2 → ()]
Possible solutions are to demand L1 to be as long or as short as possible.

Some examples of transformation rules:

```
statlist (L1: Stm*, statlist (L2: Stm*), L3: Stm*) => statlist (L1, L2, L3)
statlist (L1: Stm*, noop, L2: Stm*) => statlist (L1, L2)
statlist (S: Stm) => S
statlist () => noop
```

4.4. Unions of pattern

Suppose we want to match a value that is either a sum or a product. Introducing an 'or' operator '|' we may write:
A: (add (Exp, Exp) | mul (Exp, Exp))

The '|' operator is a potential source of non-determinism:
add (A: Exp, B: mul (_, _)) | add (B: mul (_, _), A: Exp)
applied to some values where 'mi' denote products, and 'ei' expressions not being products:

add (e1, e2) does not match and yields ∅.
add (e1, m2) matches once, yields { [A → e1; B → m2] }
add (m1, m2) matches twice, giving
{ [A → m1; B → m2] , [A → m2; B → m1] }

4.5. Intersection of patterns

An operator '&' can be imagined to denote the intersection of patterns. A pattern "p1 & p2" matches if and only if "p1" and "p2" both match and binds all names occurring in "p1" or "p2".

Assume you want to match a list of terms containing at least one term of sort S1 **and** at least one of sort S2. The solution
(_*, S1, _*, S2, _*)
forces the S1 term to occur before the S2 term. The real solution is
(_*, S1, _*) & (_*, S2, _*)

4.6. Complement patterns

Having introduced 'and' and 'or' operation, a 'not' operation may also be imagined. If "p" is a pattern matching some values, then "!p" is a pattern matching all but those values.

"A: !Stm" matches any value (tuple) but not a single statement.

"A: (_ & !Stm)" matches any term, but not a statement and not tuples.

"SL: statlist (_ & !noop)*" and
"SL: statlist ((_*) & !(_*, noop, _*))" both match any statement list not containing a "noop" statement.

Naturally, it does not make sense to put names into a pattern after '!', e.g. !(A: Stm). Thus we restrict patterns used with the complement operator '!' by forbiding them to contain names.

4.7. Name conflicts

If a name in a pattern occurs more than once not separated by '|' or occurs inside an iterated subpattern, a name conflict results: we have to define which value the name shall be bound to. Different operands of '|' are independent, thus the very last example of section (4.4) does not introduce a name conflict.

Let p = add (E: Exp, E: Exp) be a typical example for a name conflict. We want to apply p to a value v = add (a, b) where a and b are values of type Exp. Different solutions can be imagined:

- Restrictive solution: Patterns such as p are not allowed. They are much like double declarations.
- Non-deterministic solution: p applied to v returns two environments [E → a] and [E → b] i.e. one of the two possible values is selected.
- Concatenation: one environment is returned where E is bound to the pair (a, b). In general, all possible values for a name are concatenated.
- Unification: if a equals b, E is bound to a (or b), otherwise the pattern does not match.

Restrictive and non-deterministic solution are not satisfactory and need no further discussion. In the example above, unification semantics is most natural.

A pattern "L: Stm*" does not lead to a name conflict, L is bound to a list of statements. The pattern p = (L: Stm)* imposes a conflict since L is matched to all statements in turn. Applied to a value v = (s1, ..., sn), the pattern bindings L → s1, ..., L → sn are in conflict. In concatenation semantics, L is bound to the concatenation (s1, ..., sn), i.e. p is equivalent to "L: Stm*". In unification semantics, p only matches if all si are equal.

Pattern p = (while (E: Exp, Stm))* matches tuples of while statements. In concatenation semantics, E is bound to the list of all conditions of these while statements whereas with unification, p only matches lists of while statements all having the same condition and binds E to this condition. The effect of concatenation semantics may be obtained indirectly by something like

```
map (while (E: Exp, Stm) => E)    where
map (f) (a1, ..., an) = (f a1, ..., f an).
```

Unification has two great advantages: it is in most cases more intuitive than concatenation and allows for instantiating general rules to more specific ones (see section 5.3). Due to these advantages, we shall adopt unification in the sequel although there is the drawback that equality has to be defined and implemented.

4.8. Applicative patterns and tree fragments

Assume we want to write a pattern that binds a name to a constructor and other names to its operands, e.g. a pattern for sums or products:

```
(O: (add | mul)) (A, B)
```

It binds O to "add" resp. "mul" and A and B to the operands. Here we have used a new operator on patterns, the invisible application operator, and we have extended the set of values such that it contains both lists of terms and a special kind of mappings between these.

$$\text{Value} \subseteq \text{Term}* \cup (\text{Term}* \to \text{Term}*)$$

The additional values are called tree fragments or contexts, and represented as terms containing exactly one leaf "[]". This leaf may be thought of as a hole where the argument is to be inserted. (A generalization to fragments containing more than one hole may be imagined but is not described here for simplicity).

A context may be applied to any other value. This application is merely a syntactic insertion or substitution: "f v" is the value resulting from replacing the "[]" leaf of "f" by "v". If "v" is a complete term, "f v" will also be complete, if "v" is a fragment, then "f v" will be, too.

Examples for tree fragments:

add []	add (a, [])
if (c, [], e)	if (c, [])
statlist []	statlist ([], s1, s2, s3)

Examples for syntactic insertion:

(if (c, [], e)) t	= if (c, t, e)
(add []) (a, b)	= add (a, b)
(if (c, [])) (statlist [], e)	= if (c, statlist [], e)
(statlist (s1, s2, [])) (s3, s4)	= statlist (s1, s2, s3, s4)

Alternative paraphrasing: "c" instead of "c []" for any constructor c not having arity 0.
Note the differences: "true" is just the term "true" since "true" has arity 0. "statlist" or "statlist []" denote a tree fragment, but "statlist ()" denotes a term with an empty list of subterms.

Applicative patterns "p1 p2" are a potential source of non-determinism: when they are applied to a value v, v is separated in all possible ways into two values v1 and v2 such that v = v1 v2 holds. Then p1 is applied to v1 and p2 to v2, and the resulting environments are unified.

In order to obtain more powerful patterns, we introduce some more wild cards:

- "op" matches any operator except those with arity 0.
- "nullop" matches any operator with arity 0.
- "listop" matches any list operator, i.e. operator with arity *.
- "^" matches any tree fragment Term * → Term, but not Term * → Term *.
- "%" matches any arbitrary value (see also section (5.5) below)

Examples:

(O: op) (A, B)

applied to	*yields*
add (a, b)	[O → add = add []; A → a; B → b]
statlist (s1, s2)	[O → statlist = statlist []; A → s1; B → s2]
statlist (s1, s2, s3)	∅, since the list of children does not match

(O: op) (A, (O: op) (B, C))   applied to   add (a, mul (b, c))
gives ∅ due to unification semantics (conflict O → add, O → mul)

Let v = div (add (a, b), mul (c, d)).
(O: (add | mul)) (A, B)   applied to   v   gives ∅ since root does not match.

(U: ^) (O: (add | mul)) (A, B)   matches v in two ways:
[U → div ([], mul (c, d)); O → add; A → a; B → b] and
[U → div (add (a, b), []); O → mul; A → c; B → d].

Alternative paraphrasing: "(N: ^) p" where N is a name and p a pattern may be abbreviated to "N p". If no pattern follows, "N: ^" must not be abbreviated since "N" means "N: _".

5. The transformation language

5.1. Objects of the language

The transformation language shall be a tool to manipulate some kinds of objects. First class objects are the values described above: simple terms, tuples of terms, term fragments, and tuples with exactly one fragment component. They may be analyzed by patterns and synthesized by expressions without restriction.

Second class objects are all kinds of functions between objects that are not term fragments. They must not be subject of pattern matching, but may only be defined and applied.

5.2. Syntax and semantics of the language

Basic syntactic sorts of the language are definitions, rules, patterns, and expressions. Syntax and semantics of patterns are already described in the previous chapter. A rule is written as

$$p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n$$

where all p_i are patterns and e_i expressions. It denotes a non-deterministic mapping of first class objects. Applied to a value v, the patterns p_i produce sets of environments se_i. Each expression e_i is evaluated in any environment contained in se_i to values forming a set sv_i of values. The set of values produced by the whole rule applied to v is then the union of the sets sv_i. On the rule level, non-determinism may be eliminated by the convention that the leftmost matching partial rule wins.

Expressions are built up from names by concatenation (comma) and application. Concatenation is only allowed for first class objects. There are different kinds of application:

- application of a first class object to another; it is merely a syntactic insertion,
- application of a second class object denoted by its name to some values,
- application of an unnamed rule to some value

For the latter application we propose some ways of alternative paraphrasing.

Original form:	Paraphrasing:
{p1 => e1 | .. | pn => en} e	case e of {p1 => e1 | .. | pn => en}
{N: % => e} e’	let N = e’ in e
{p => e} e’	let p match e’ in e
let .. in let .. in	let .. and .. in

A named function may be defined by

Name (formal parameters) = Functional-expression

A functional expression is built up by application of functions of higher order to other functions or rules.

5.3. Examples

Function to simplify "if" terms:

```
sim-if =        { if (true, T, _) => T
                | if (false, _, E) => E }
```

another solution: sim-if = { if ((true, S, _) | (false, _, S)) => S }

assoc (O) = { (O: op) (A, O (B, C)) => O (O (A, B), C) }

The function "assoc" maps nested "O" terms into their left associative form assuming O is an associative binary operator. It may be instantiated e.g. "assoc (add)" or "assoc (sep)". Note that instantiation crucially depends on unification semantics: when the root of the input term is considered, O is already bound to the actual parameter and the rule matches only if the root is labeled by this parameter.

Parameters of function definitions may be functions:

comp (F, G) = { X: % => G (F (X)) }

"comp (F, G)" is written as "F o G" in the sequel (to be read "first F, then G").

Function definitions may be recursive:

```
map (F) =    { () => ()
             | (H: _, T: _*) => (F H, map (F) T) }
```

Note that this function only works with lists of terms, not lists of term fragments; otherwise we would have to write "(_ | ^)" instead of "_". There is a slight difference between our "map" and the "map" of other functional languages due to the meaning of comma as concatenation:

```
double = { X: _* => (X, X) }
double (a1, a2, a3) = (a1, a2, a3, a1, a2, a3)
map (double) (a1, a2, a3) = (a1, a1, a2, a2, a3, a3)
```

More useful functions may be defined if we introduce a built-in sort "Nat" with elements 0, 1, 2 etc. and standard functions +, - etc.

```
times =    { (0, _*) => ()
           | (i: Nat & !0, X: _*) => (X, times (i-1, X)) }
```

concatenates a list i times with itself. If '|' is made deterministic, "Nat & !0" may be simplified to "Nat".

5.4. Example: translation between grammar 1 and grammar 2

To translate terms of grammar 1 into grammar 2, we must transform "sep (a, b)" constructs into "statlist (a, b)" and concatenate statement lists resulting from nested "sep" terms. At first an auxiliary mapping:

```
strip =    { statlist (L: Stm*) => L
           | S: (Stm & !statlist (_*)) => S }
```

It strips the operator "statlist" from statement lists leaving the tuple of children. With deterministic semantics, the second part could have been written simply "S: Stm => S".

```
tr1to2 =
      { O: nullop => O  /* nullary operators are not affected */
      | (O: op & !sep) (L: _*) => O (map (tr1to2) L)
            /* no constructs besides "sep" are affected,
               the translation is applied recursively to the children */
      | sep (S1: Stm, S2: Stm) =>
            statlist (strip (tr1to2 S1), strip (tr1to2 S2) )
            /* the children are translated and nested "statlist" terms
               are avoided by "strip" */  }
```

Terms of grammar 2 are translated into grammar 1 by transforming "statlist" terms into left associative nested "sep" terms.

```
tr2to1 =     { O: nullop => O
             | (O: op & !statlist) (L: _*) => O (map (tr2to1) L)
             | statlist () => noop
             | statlist (S: Stm) => tr2to1 S
             | statlist (L: Stm+, S: Stm) => sep (tr2to1 (statlist L), tr2to1 S) }
```

The operator "statlist" may just be omitted if the list consists of at most one statement. The case of one element is also the end of the recursion over statement lists. The last case matches only lists consisting of at least two elements. The rightmost statement is taken out of the list.

Note that both translations are deterministic and work for all terms (not tuples and not term fragments) of the respective grammar i.e. they never fail if they are applied to suitable values. The auxiliary mapping "strip" leads to the question of modules or locally defined functions.

5.5. Recognition of failures

If we make patterns and rules deterministic by some decision strategy, there are only two possible kinds of results when a rule is applied to a value v: either a new value v' or failure. To catch the failure result, we assume that there are ordinary values as described until now and an additional value "fail". Concatenation and syntactic insertion of values should be strict, i.e.

```
(fail, v) = (v, fail) = fail;  fail v = v fail = fail.
```

Assuming the wild card '%' to match any value except "fail" we can write down useful functions of higher order:

```
try (T)        = { X: % =>
                    case T X of
                        { Y: % => Y
                        | fail => X } }

repeat (T)     = { X: % =>
                    case T X of
                        { Y: % => repeat (T) Y
                        | fail => X } }
```

"try (T)" tries to apply the rule T to some input value X. If T matches X, the result of applying T is returned, otherwise the original value X is taken as result. This is similar to the semantics of rules in current OPTRAN. "repeat (T)" applies rule T repeatedly until it does not longer match.

The curly brackets {} are used to resolve syntactic ambiguities on the rule level, and to delimit the scope of identifiers, i.e. Y is local to the last block, but it may be used as well as the more global T and X.

The semantics of the set of rules r1, ..., rn of the current OPTRAN system may be expressed in functional OPTRAN by "repeat (r1' | ... | rn')" where ri' results from ri by changing the syntax and providing an upper environment, e.g.

```
< sep , A , < sep , B , C > > into < sep , < sep , A , B > , C >
```

must be changed to

```
(U: ^) sep (A, sep (B, C)) => U sep (sep (A, B), C)
```

5.6. Global strategies

Global strategies may be expressed by functions of higher order: "topdown (T)" applies T at each subterm, then visits recursively the children of the subterm. "bottomup (T)" applies the transformation T first at the children, then at the parent. A combination of both strategies applying some T1 top down and some T2 bottom up could be imagined.

```
topdown (T) =      try (T)
                   o  { O: nullop => O
                      | (O: op) (L: _*) => O ( map (topdown (T)) L ) }

bottomup (T) =     { O: nullop => O
                      | (O : op) (L: _*) => O ( map (bottomup (T)) L ) }
                   o  try (T)
```

The argument transformations are called by the strategy via "try" to avoid the burden of writing patterns covering all cases in the definition of T.

Example:

```
atomic-1to2 = { sep (S1: Stm, S2: Stm) => statlist (strip S1, strip S2) }
tr1to2 = bottomup (atomic-1to2)
```

There is a drawback of both "topdown" and "bottomup": the input terms of mappings such as "atomic-1to2" are mixed between two languages: the root is not yet translated e.g. "sep" is an operator of grammar 1, but the children are, i.e. "Stm" is the sort "Stm" of grammar 2. If our two example grammars differed more, we would not be able to use the "bottomup" strategy. To overcome this difficulty we propose another form of "bottomup" where tuples instead of terms are transported:

```
bottomup (T) =          { O: nullop => O
                             | (O : op) (L: _*) => (O, map (bottomup (T)) L ) }
                        o   try (T)
```

Then "atomic-1to2" must be changed, too:

```
atomic-1to2 = { (sep, S1: Stm, S2: Stm) => statlist (strip S1, strip S2) }

atomic-2to1 =
        { (statlist) => noop
        | (statlist, S: Stm) => S
        | (statlist, L: Stm+, S: Stm) => sep (atomic-2to1 (statlist, L), S) }
```

Note that "(statlist)" in the first case is the product of the recursive bottom up call applied to "statlist ()". The enclosing parentheses are superfluous in this pattern.

The notation using the bottom up scheme is more comfortable, more compact and concentrates on the essential parts. In the translation from grammar 2 to grammar 1, the difference between vertical recursion hidden in "bottomup" and horizontal recursion remaining in "atomic-2to1" becomes visible.

5.7. Preventing concatenation

Assume there is a function producing two elements A and B of Stm*. The notation developed so far only allows for returning the concatenation (A, B) of both elements. If they are used as input of another function, their original values cannot be retrieved by a pattern since there is no criterion to decide where A ends and B begins.

An ugly solution would be to insert a dummy value, e.g. an element of Exp, to separate both tuples. Another solution would be to group the statement tuples by use of a constructor of the object language, e.g (statlist A, statlist B).

We propose a general mapping Term * $\rightarrow$ Term described by angular brackets to free the user from the burden to look for suitable constructors in the object language. The pseudo constructor "<..>" may be used as any ordinary constructor in patterns and expressions.

The function mentioned above may then return (<A>, <B>) and a pattern (<A: Stm*>, <B: Stm*>) may then be used to retrieve the original values of A and B.

Example of a function transforming a list of pairs into a pair of lists:

```
transpose =       { () => ()
                  | ( < <H1: %>, <H2: %> >,   T: % ) =>
                       let ( <L1: %>, <L2: %> ) match transpose T
                           in ( <H1, L1>, <H2, L2> ) }
```

5.8. Interactive OPTRAN

In interactive OPTRAN, we need mechanisms to designate a subterm of an enclosing term as "actual" or "current" term. The user may then walk through the enclosing term i.e. designate another subterm to be the actual one, and apply a transformation to the current subterm. The root of the actual subterm is called the actual node.

Terms with designated subterm may be described by a pair (u, s) where s is the designated subtree and u its upper environment (a term fragment) such that "u s" is the enclosing term.

Examples for useful transformations:

```
up = { ( (U: ^) (O: op) (L: _*, [], R: _*) , A: _? ) =>
                  ( U , O (L, A, R) ) }
```

This transformation is a motion from the actual node to its parent. The actual subterm may be a real term or an empty tuple "()" thus the type for it is "_?". The parent O, the left siblings L, the right siblings R, and the actual subtree A form the new actual subtree O (L, A, R). The transformation fails if the actual node is the root of the input tree, since the input looks then like ([], A) and the subpattern (O: op) does not match.

Pairs "(u, s)" of a subterm and its upper environment could be written more readably by alternative paraphrasing as "u [s]" e.g.

```
up = { U (O: op) (L: _*, [A: _?], R: _*)   =>   U [ O (L, A, R) ] }
```

The square brackets then denote the actual subterm.

Other motions and transformations:

```
right = { U (L: _*, [A: _?], B, R: _*)   =>   U (L, A, [B], R) }
```

It fails if we are at a rightmost child.

```
child1 =     { U [ (O: op) (A: _, R: _*) ]   =>   U O ([A], R)
             | U [ (O: listop) () ]   =>   U O [()] }
```

It fails if we are positioned at a nullary operator. List operators having at least one child are treated by the first case.

```
toroot  =  { U [A: _?] => [U A] }  =  repeat (up)
apply (T)  =  { U [A: _?] => U [T A] }
```

"apply (T)" applies the transformation T to the actual subtree.

```
remove = { U (O: listop) (L: _*, [_?], R: _*)   =>   U O (L, [()], R) }
```

This rule removes the actual subtree and leaves the "cursor" at a place between two adjacent subtrees. The next "right" or "left" command will move the "cursor" to one of these subtrees.

The reader is invited to imagine further useful motions and transformations.

6. A complex transformation: removal of function calls

6.1. The problem

At last, we shall give a larger example of program transformation. The problem is to replace function calls in a Pascal like language by procedure calls; it is described by Qian Zhenyu in [3].

The abstract syntax as described by Qian is given in a notation where most constructors are distfix or even unvisible. We think that this syntax is suitable for a user interface, but not for defining a transformation. Thus we shall present the syntax in our form where most direct recursions are avoided by iterators.

We shall use capitalized words for syntactic sorts, completely upper case words for parameters in patterns, and lower case words for constructors and names of transformation rules.

6.2. Abstract syntax

```
Program       ::= Proc-dec
Proc-dec      ::= procedure (Id, Pm-list, Decl-part, Body)
Func-dec      ::= function (Id, Pm-list, Decl-part, Body)
Decl-part     ::= decllist Decl*
Decl          ::= Proc-dec | Func-dec
Pm-list       ::= pmlist Id*
Body          ::= Stm-list
Stm-list      ::= statlist Stm*
Stm           ::= assign (Id, Exp) | pcall (Id, Exp-list)
                  | if (Exp, Stm-list, Stm-list) | while (Exp, Stm-list)
                  /* added to make things more interesting */
Exp-list      ::= explist Exp*
Exp           ::= Bin (Exp, Exp) | Un (Exp) | Id | Constant
                  | fcall (Id, Exp-list)
Bin           ::= add | sub | ...
```

```
Un          ::= sign | ...
Constant    ::= 0 | 1 | ...
Id          ::= ...
```

6.3. Subtasks to be done

The global task may be divided into several subtasks:

1) Function calls must be extracted from expressions
2) They must be transformed into procedure calls
3) Function declarations must be changed into procedure declarations

A serious problem is the production of new names for procedures created by (3) - the function name is used as additional parameter - and for new variables introduced by (1). We assume two functions "functoproc: Id → Id" and "newid: → Id" for simplicity. Naturally both functions depend on the set of names used in the program to be transformed, and on the names produced by previous calls of them.

6.4. Bottom up solution

We want to use the bottom up scheme described earlier:

```
trafo = { P: Program => bottomup (tr) P }
```

The recursive calls of "bottomup (tr)" translate expressions (members of Exp) and expression lists (members of Exp-list) into the form "< <SL>, <E> >" where "E" is the expression list without function calls and "SL", an element of Stm*, is the list of extracted calls, already translated into procedure calls.

```
tr = { lettype Transexp = < <Stm*>, <Exp | Exp-list> > in ...
```

The "lettype" construct introduces an abbreviation for a type (a pattern not containing any parameter name).

```
( O: (Bin | Un | explist), TEL: Transexp* ) =>
   let ( <SL: Stm*> , <EL: Exp*> ) match transpose TEL
      in < <SL> , < O (EL) > >   | ...
```

This partial rule is a little bit tricky: a term "o (e1, ..., en)" where o ∈ Bin ∪ Un ∪ {explist} and ei ∈ Exp is already translated by the recursive call into "(o, te1, ..., ten)" where tei ∈ Transexp. "transpose" transforms the list "(te1, ..., ten)" of pairs into a pair of lists: "(<sl> , <el>)" where "sl" is the concatenation of all lists of calls contained in the "tei" values, and "el" is a list of n expressions corresponding to "(e1, ..., en)" but without function calls. At last, a new element of "Transexp" is built by these values.

```
( fcall, FI: Id, < <PL: Stm*> , <explist (EL: Exp*)> > ) =>
   let  NI = newid ()
   and  PI = functoproc FI
   in < <PL, pcall (PI, explist (EL, NI))> , <NI> >   | ...
```

Input of this partial rule is a function call whose actual parameter list is already translated into a member of "Transexp". FI is the function name, PL the list of calls extracted from the actual parameter list, and "explist EL" the parameter list without calls. The function call is replaced by a new identifier NI. The variable NI is set by a procedure call where NI is an additional actual parameter; the corresponding formal one is FI as we shall see below. The new procedure call is appended to the list PL of extracted calls. The resulting object is again an element of "Transexp".

The translation of expressions is now completely defined - Id's and Constant's are not affected - and we continue by defining how the translated expressions are to be inserted into their context.

```
( assign, AI: Id,
            < <pcall (PI: Id, explist (EL: Exp*, NI: Id))> , <NI: Id> > )
  => pcall (PI, explist (EL, AI))   | ...
```

This partial rule serves only to optimize the transformation and could be omitted. It matches if there is only one extracted call and the expression consists of a single variable i.e. if the assignment had the form "assign

(AI, fcall (FI, explist EL))" before the expression was translated by the recursive call of "bottomup (tr)". We assume deterministic rules such that the partial rule above always overrides the following more general rule:

```
( assign, AI: Id, < <PL: Stm*> , <E: Exp> > )
  => (PL, assign (AI, E))   | ...
```

The list of extracted calls is simply placed to the left of the assignment. Remember that this rule is called by the bottom up scheme in the form "map (bottomup (tr)) sl" where sl is in Stm*. The meaning of comma as concatenation - compare with "map (double)" - suffices to insert PL correctly in its context.

```
( if, < <PL: Stm*> , <E: Exp> >, SLL: Stm-list {2} )
  => (PL, if (E, SLL))   |

( pcall, I: Id, < <PL: Stm*> , <EL: Exp-list> > )
  => (PL, pcall (I, EL)   |

( while, < <PL: Stm*> , <E: Exp> >, statlist (SL: Stm*) )
  => (PL, while (E, statlist (SL, PL)))   | ...
```

The rules for "if" terms and procedure calls are analogous to that for assignments, but in loops, the list of extracted calls must be doubled: it must be inserted once in front of the loop and once at the end of its body.

```
( function, FI: Id, pmlist (PL: Id*), D: Decl-part, B: Body )
  => procedure (functoproc FI, pmlist (PL, FI), D, B)
}            /* end of "tr" */
```

This is the end of the definition of "tr". Note that "D" and "B" are already transformed if this part is applied. It changes the name into a procedure name and appends the function name as last formal parameter to the parameter list.

6.5. Repetitive solution

A quite different solution is obtained by using the "repeat" scheme. Whereas the bottom up scheme explicitly describes the order of rule application, the following solution does not specify it. Thus it cannot explicitly described that e.g. the calls extracted from a left summand occur to the left of the calls extracted from the right summand. The bottom up solution is finer and thus more tedious to be specified; it carries the extracted calls step by step out of an expresseion whereas the following solution transports them by one big step.

Assume there are transformations "simpleassign" to transform assignments whose right hand side is a single function call, "extract" to extract calls from expressions, and "decl" to change declarations.

```
rs    = repeat (simpleassign)
re    = repeat (extract)
rd    = repeat (decl)

rsd   = repeat (simpleassign | decl)
red   = repeat (extract | decl)
```

Then "trafo" might be specified in many several ways (remember f o g means "first f, then g"):

```
rs o red,   rsd o re,   rs o re o rd,   rs o rd o re,   rd o rs o re
```

The different partial rules may be applied in any order, only "simpleassign" must be applied before "extract" since "extract" is more general.

```
simpleassign =
      { U assign (AI: Id, fcall (FI: Id, explist (EL: Exp*)))  =>
            let PI = functoproc FI
            in  U pcall (PI, explist (EL, AI)) }
```

The presence of the upper environment variable "U" makes the rule non-deterministic. It may be applied as long as there are assigned function calls somewhere in the tree. But "rs" is deterministic in its result although each step done by it is not deterministic since the final result of "rs" does not depend on the order of transformation of individual assignments.

```
decl = { U function (FI: Id, explist (EL: Exp*))  =>
                 let PI = functoproc FI
                 in  U procedure (PI, explist (EL, FI)) }
```

This rule transforms an occurrence of a function declaration into a procedure declaration.

```
extract = { U  (S: (if ([], Stm-list {2})
                        | pcall (Id, explist (Exp*, [], Exp*))
                        | assign (Id, []) ))
                   E  fcall (FI: Id, explist (EL: Exp*))   =>
            let    NI = newid ()
            and    PI = functoproc FI
            and    PC = pcall (PI, explist (EL, NI))
            in     U (PC, S E NI)
          | U  while ([], statlist (SL: Stm*))
                   E  fcall (FI: Id, explist (EL: Exp*))   =>
            let    NI = newid ()
            and    PI = functoproc FI
            and    PC = pcall (PI, explist (EL, NI))
            in     U (PC, while (E NI, statlist (SL, PC)) }
```

This huge transformation rule extracts function calls out of expressions. The first eight lines treat expressions not occurring as condition of a "while" statement. The first pattern separates the input tree into four tree fragments: the upper environment U that is not affected by the transformation, a statement fragment S as context of some expression, an expression fragment E surrounding a function call. This function call is transformed into a procedure call PC and a new variable NI is introduced. Then the result tree is formed of U where the occurrence of S is replaced by the pair (PC, S E NI). E NI is the original expression where the function call is replaced by NI. Informally, the essence of this part could be written as

```
U S E fcall (..) => U (pcall (.., NI), S E NI)
```

The second part deals with "while" statements, it cannot be combined with the first part since the extracted call must be placed at the end of the body of the loop, too.

The task of transforming a function call into a procedure call may be abstracted out of "extract":

```
trafocall = { fcall (FI: Id, explist (EL: Exp*))   =>
                let    NI = newid ()
                and    PI = functoproc FI
                and    PC = pcall (PI, explist (EL, NI))
                in     (PC, NI) }

extract = { U  (S: (if ([], Stm-list {2})
                        | pcall (Id, explist (Exp*, [], Exp*))
                        | assign (Id, []) ))
                   E  (FC: fcall (_*))   =>
            let    (PC, NI)  match  trafocall FC
            in     U (PC, S E NI)
          | U  while ([], statlist (SL: Stm*))
                   E  (FC: fcall (_*))   =>
            let    (PC, NI)  match  trafocall FC
            in     U (PC, while (E NI, statlist (SL, PC)) }
```

References

[1] Badt, Möncke, Raber: Specification of Recursive Patterns
[S.1.6-SN-4.0]

[2] Miranda: A non-strict functional language with polymorphic types,
Springer Lecture Notes in Computer Science, vol. 201

[3] Qian, Zhenyu, An Example of Recursively Presented Transformation
[M.1.1.S1-SN-21.0]