A Functional Language for the Specification of Complex Tree Transformations

Reinhold Heckmann

FB 10 - Informatik
Universität des Saarlandes
6600 Saarbrücken
Bundesrepublik Deutschland

email: heckmann@cs.uni-sb.de

ABSTRACT

Transformations of trees and rewriting of terms can be found in various settings e.g. transformations of abstract syntax trees in compiler construction and program synthesis.

A language is proposed combining features of a general purpose functional language with special means to specify tree transformations. Atomic transformations are considered first order functions and described by pattern matching. The pattern specification language allows for partitioning trees by arbitrary vertical and horizontal cuts. This goes beyond what is possible in similar languages [2,13,14]. High order functions and functional combinators are used to express strategies for the controlled application of transformations.

1. Introduction

The PROSPECTRA project (PROgram development by SPECification and TRAnsformation) aims to provide a rigorous methodology for developing **correct** software [10]. It integrates program construction and verification, and is based on previous work in the CIP project [3].

A formal specification is gradually transformed into an optimized executable program by stepwise application of transformation rules. These are carried out by the system, with interactive guidance by the implementor, or automatically by transformation tools.

Each transition from one program version to another is done by application of an individual transformation rule or a transformation script invoking rules systematically. The language to express such transformation scripts as well as individual rules is designed in functional style since transformations are functions in some tree domain, and scripts may be build up by calling high order functions parametrized by transformations.

Starting points for the transformation language ′TrafoLa′ were the functional languages HOPE [2], ML [13], and Miranda

---

This research was partially supported by the Commission of the European Communities under Esprit Project Ref. No. 390 (PROSPECTRA) and by Sonderforschungsbereich 124 - VLSI Design Methods and Parallelism

[14] which are quite similar with respect to their functional kernel. They all use a very restricted form of pattern matching for trees that only allows for the specification of some fixed region near the root of the tree and for selecting subtrees adjacent to this region. Thus, it is neither possible to specify a region or to refer to a subtree whose root is far from the root of the whole tree, nor to bind the context of such a subtree to a variable.

In these languages, sequences are represented as trees, and thus, their treatment is always biased to their leftmost item. Patterns only allow for selecting a fixed number of items at the left end of the sequence and its remainder. It is impossible to access some infix or the last item of a sequence directly.

A completely different approach was given by Huet [9]. His second order patterns allow for the specification and selection of arbitrary subtrees of a given subject tree. The patterns are used to express powerful transformations, but they are not embedded into a functional language. Pattern matches usually result in more than one solution, but there is no means to control sets of solutions by ′Boolean′ pattern operators. In fact, Huet's patterns contain only a few pattern operators, and one - the inverse application - is extremely complex. The generality of this operator is often not needed, and consequently, TrafoLa provides other, less complex operators for usual purposes.

TrafoLa was developed by increasing the power of patterns of Hope [2], ML [13], and Miranda [14] towards the power of Huet's patterns [9] - or in other words, by embedding Huet's patterns into a nice functional language. Non-determinism is used to ease the description - the user only needs to specify the shape of the subtree he/she wants to select, not where or how such a subtree is to be found (nevertheless, this is also possible). The resulting set of solutions is handled following Prolog [4] - by enumerating the elements by means of backtracking or by cutting it down to one element.

We shall first consider the structure of the objects to be transformed. Then we shall define patterns and raise their power step by step. At last, they will allow for partitioning trees by arbitrary vertical and horizontal cuts. We shall introduce functions describing transformations and functional combinators. Finally, we shall give an example of a complex transformation. We do not treat the development or correctness of transformations; this is done in [12].

2. Objects of the transformation language

The objects of the transformation language called ′values′ are trees and sequences of trees - the objects to be transformed - and also functions - these are transformations and strategies.

The domain of values is recursively defined as follows:

1) Constants such as ′true′ and ′0′ are values.
2) There is a special constant ′@′ indicating the place where a subtree was cut out of some bigger tree.
3) Any sequence ′[v_1, ... , v_n]′ of values is a value. A special case of this is the empty sequence ′[]′. We do not distinguish between tuples and sequences.
4) Each tree is a value. A tree ′op v′ consists of a root operator ′op′ and some value ′v′ standing for the children list of the tree.
5) Functions mapping values into values (or sets of values) are functional values.

Examples of values: 0 [] [1, 2, 3]

ife [eq [i1, i2], add [i1, 1], sub [i1, i2]]

where ′0′, ′1′, ′2′, ′3′, ′i1′, and ′i2′ are constants, and ′ife′, ′eq′, ′add′, and ′sub′ are operators.

Values may be checked for equality and inequality. For functional values, this is realized by some approximation e.g. syntactic comparison, guaranteeing that different functions are never claimed to be equal - whereas equal functions may be claimed to be different.

We adopted the following conventions in TrafoLa:

1) Square brackets [] are used as tuple and sequence delimiters. Parentheses () will be used in patterns and expressions of TrafoLa only to solve syntactic ambiguities.

2) Constant and operator names start with a lower case letter or are numbers.

3) Variable names (they did not yet occur) start with upper case letters.

4) Type names are printed in *italics*, and keywords of TrafoLa in **bold face**.

Subsets of values may be specified by data type definitions. We give an example describing the abstract syntax of a tiny imperative programming language. The generated trees will also be the objects of our example transformations.

type *Program* = *Proc-dec*

and *Proc-dec* = procedure [*Id*, *Id**, *Decl**, *Stm**]

and *Func-dec* = function [*Id*, *Id**, *Decl**, *Stm**]

and *Decl* = *Proc-dec* | *Func-dec*

and *Stm* = noop | assign [*Id*, *Exp*] | pcall [*Id*, *Exp**] |
ifs [*Exp*, *Stm**, *Stm**] | while [*Exp*, *Stm**]

and *Exp* = false | true | *Num* | *Id* | *Unop* [*Exp*] |
Binop [*Exp*, *Exp*] | ife [*Exp*, *Exp*, *Exp*] | *Fcall*

and *Fcall* = fcall [*Id*, *Exp**]

and *Num* = 0 | 1 | 2 | etc.

and *Id* = ... (identifiers)

and *Unop* = sign | not

and *Binop* = eq | lt | add | sub | mul | etc.

The construct *'T*'* denotes the type of all sequences of items of type *'T'*.

Later, it will turn out that the type expressions at the right hand sides of the type definitions are nothing else than patterns, i.e. data type definitions may be viewed as recursive patterns.

Example of a value with type *'Stm'*:

while [lt [i, n] , [assign [s, add [s, i]], assign [i, add [i, 1]]]]
/* while i < n do s := s + i; i := i + 1 od */

3. Patterns

3.1. Informal semantics of patterns

The following transformations seem to be useful:

dec If-true	= { ife [true, T, E]	=>	T }
dec If-false	= { ife [false, T, E]	=>	E }
dec While-false	= { while [false, S]	=>	noop }

These are declarations binding functional values - denoted by the construct ′{ pattern => expression }′ - to the variables ′If-true′ etc.

What is the semantics of the function ′{p => e}′ applied to some value *v*?

If *p* does not match *v*, the rule fails, otherwise the variable names occurring in *p* are bound to values (subterms of *v*). Thus *p* matched against *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. ′If-true′, for instance, is undefined for ′while′ statements, even when a matching ′if′ expression occurs in its condition or body.

Later, we shall consider non-deterministic patterns that may match in different ways thus returning a set *s* of environments when matched against a value *v*. The failure case fits well with this view, the pattern then returns the empty set of environments. The expression *e* will be evaluated in *s*, and thus eventually produce a set of results. Later, we shall present different methods how to handle this non-determinism.

3.2. Formal semantics of patterns

The formal semantics of patterns is described by means of a semantic function

$$P:\ \text{Pattern} \rightarrow \text{Env} \rightarrow \text{Value} \rightarrow 2^{\text{Env}}$$

matching a pattern against a value in some environment and producing a set of environments. An environment is a mapping from variables to values or ′unbound′. We shall denote environments by

$$<A_1 \rightarrow v_1; \ldots ; A_n \rightarrow v_n>$$

were A_i are distinct variables and v_i values (not ′unbound′).

Notice that we shall abstract from error cases when we shall present parts of the definition of P.

3.3. Atomic patterns

Atomic patterns are constructors, variables, syntactic types, and wild cards.

A constructor (constant or operator) *c* (or @) matches just itself:

$$P\,(c)\ r\ v \ =\ \text{if}\ \ v = c\ \ \text{then}\ \ \{<>\}\ \ \text{else}\ \ \emptyset$$

If the value equals *c*, the match succeeds returning just one environment, namely <> i.e. the empty environment mapping all variables to ′unbound′. Otherwise, the match fails returning ∅, the empty set of environments.

Variables may be used in two ways: either to bind subvalues or to import values into a match.

Binding variables (called ′open′ in [1]) match any value and create a new environment where they are bound to this value:

$$P\ (A)\ r\ v \ = \ \{\langle A \rightarrow v \rangle\}$$

Importing variables (called 'closed' in [1]) match just the value they are bound to in the environment of the match:

$$P\ (\%\ A)\ r\ v \ = \ \text{if}\ \ v = r\ (A)\ \ \text{then}\ \ \{\langle\rangle\}\ \ \text{else}\ \ \varnothing$$

When a type name such as *'Stm'* is encountered in a pattern, its meaning is looked up in the environment of the match. The meaning will be a predicate on values.

$$P\ (T)\ r\ v \ = \ \text{if}\ \ r\ (T)\ v\ \ \text{then}\ \ \{\langle\rangle\}\ \ \text{else}\ \ \varnothing$$

The wild card '_' matches any value: $P\ (\_)\ r\ v \ = \ \{\langle\rangle\}$

Constructors, binding variables, and the wild card also occur in Hope, ML, and Miranda, but importing variables and types don't.

3.4. Structural patterns

Structural patterns specify the structure of the matched value. They consist of subpatterns to match designated subvalues, and the resulting sets of environments are combined into one.

Sequence enumeration

The pattern $'[p_1, \ldots, p_n]'$ matches values of shape $'[v_1, \ldots, v_n]'$:

$$P\ ([p_1, \ldots, p_n])\ r\ v \ = \ \text{if}\ \ v = [v_1, \ldots, v_n]\ \ \text{then}\ \ P\ (p_1)\ r\ v_1 \oplus \ldots \oplus P\ (p_n)\ r\ v_n\ \ \text{else}\ \ \varnothing$$

If $n = 0$, this becomes to $P\ ([])\ r\ v \ = \ \text{if}\ \ v = []\ \ \text{then}\ \ \{\langle\rangle\}\ \ \text{else}\ \ \varnothing$

The combination '$\oplus$' will be defined in the next but one section. As first approximation assume that it superposes all environments in its first argument with all in its second one.

Uniform sequence

The pattern 'p*' matches sequences of arbitrary length whose items are all matched by *p*, and 'p+' matches the same sequences except the empty one '[]'.

$$\begin{aligned} P\ (p^*)\ r\ v \ = \ & \text{if}\ \ v = [v_1, \ldots, v_n], n > 0\ \ \text{then}\ \ P\ (p)\ r\ v_1 \oplus \ldots \oplus P\ (p)\ r\ v_n\ \ \text{else} \\ & \text{if}\ \ v = []\ \ \text{then}\ \ \{\langle\rangle\}\ \ \text{else}\ \ \varnothing \\ P\ (p+)\ r\ v \ = \ & \text{if}\ \ v = [v_1, \ldots, v_n], n > 0\ \ \text{then}\ \ P\ (p)\ r\ v_1 \oplus \ldots \oplus P\ (p)\ r\ v_n\ \ \text{else}\ \ \varnothing \end{aligned}$$

Tree pattern

Pattern 'p q' matches trees whose operator is matched by *p* and whose children list is matched by *q*.

$$P\ (p\ q)\ r\ v \ = \ \text{if}\ \ v = op\ w\ \ \text{then}\ \ P\ (p)\ r\ op \oplus P\ (q)\ r\ w\ \ \text{else}\ \ \varnothing$$

Note that *p* is not necessarily an operator name; this goes beyond what is possible in Hope, ML, and Miranda.

Examples

′ife [true, T, E]′ matches ′if′ expressions with condition ′true′ and binds T and E to ′then′ resp. ′else′ part.

′ife A′ matches any ′if′ expression and binds A to its children list, i.e. A is bound to a value of type [*Exp, Exp, Exp*].

′O A′ matches any tree and binds O to the operator and A to the children list.

′T′ matches any value and binds T to it.

If we don't want to bind a subvalue to a name, we may use the symbol ′_′ or the name of a syntactic sort such as ′*Stm*′.

′ifs [C, *Stm, Stm*]′ is equivalent to ′ifs [C, _, _]′ due to the structure of the language. Both patterns match ′if′ statements and bind the condition to C.

3.5. Non-linear patterns

A pattern is non-linear if a variable occurs more than once in it or occurs inside an iterated subpattern ′p*′ or ′p+′.

Let p = add [E, E] be a typical example for a non-linear pattern. *p* matches the value ′add [a, b]′ iff the subvalues *a* and *b* are equal, *E* is then bound to *a*.

Examples:

Pattern	Value	Result
add [E, E]	add [1, 1]	$\{<E \rightarrow 1>\}$
add [E, E]	add [1, 2]	$\emptyset$
A+	[1, 1, 2]	$\emptyset$
A+	[1, 1, 1]	$\{<A \rightarrow 1>\}$

Non-linear patterns are allowed in Miranda, Prolog, and Huet's language, but forbidden in Hope and ML.

3.6. Combination of sets of environments

Now, we shall define the combination ′s ⊕ t′ of two sets of environments *s* and *t*. The result is the set of all pairwise superpositions of environments where the case of inconsistent bindings of variables must be excluded in order to achieve the desired semantics of non-linear patterns:

Definition:

$s \oplus t \quad = \{ a + b \mid a \text{ in } s, b \text{ in } t, a \text{ and } b \text{ are consistent} \}$

$a + b \quad = \lambda N.\ \text{if } b(N) = \text{unbound then } a(N) \text{ else } b(N)$

a and b are consistent iff

for all variables N, a (N) = unbound or b (N) = unbound or a (N) = b (N)

Examples: Remember P ([p, q]) r [u, v] = P (p) r u ⊕ P (q) r v

Pattern	Value	Result	
[A, B]	[1, 2]	$\{<A \rightarrow 1>\} \oplus \{<B \rightarrow 2>\}$	$= \{<A \rightarrow 1; B \rightarrow 2>\}$
[A, 2]	[1, 2]	$\{<A \rightarrow 1>\} \oplus \{<>\}$	$= \{<A \rightarrow 1>\}$
[A, 1]	[1, 2]	$\{<A \rightarrow 1>\} \oplus \{\}$	$= \{\}$
[A, A]	[1, 1]	$\{<A \rightarrow 1>\} \oplus \{<A \rightarrow 1>\}$	$= \{<A \rightarrow 1>\}$
[A, A]	[1, 2]	$\{<A \rightarrow 1>\} \oplus \{<A \rightarrow 2>\}$	$= \{\}$

The superposition of environments ′+′ is not commutative - the second operand dominates the first one - , but associative, and has a neutral element, namely the empty environment ′$\langle\rangle$′. Since a + b = b + a holds iff a and b are consistent, the combination ′⊕′ is commutative, associative, and has neutral element $\{\langle\rangle\}$. In addition, it distributes over set union and satisfies s ⊕ ∅ = s. Unfortunately, it is not idempotent, e.g.

$$\{\langle A \to 1\rangle, \langle B \to 2\rangle\} \oplus \{\langle A \to 1\rangle, \langle B \to 2\rangle\} \; = \; \{ \langle A \to 1\rangle, \; \langle B \to 2\rangle, \; \langle A \to 1; B \to 2\rangle \}$$

But if the environments contained in *s* are ′uniform′ i.e. each environment binds the same set of variables, then s ⊕ s = s holds.

The algebraic properties mentioned here are stated and proved in [7] together with some additional ones.

3.7. Correspondence between TrafoLa patterns and TrafoLa expressions

Some pattern operators directly correspond to operators in TrafoLa expressions. There are expressions being constants denoting themselves, and being variables denoting the value the variable is bound to, and there are expressions ′[e_1, ... , e_n]′ for sequences and expressions ′e e’′ for trees.

The meanings of the pattern ′[A, B]′ and the expression ′[A, B]′ are inverse: the pattern ′[A, B]′ matches pairs, decomposes them into their two components, and binds A to the first one and B to the second one. The expression ′[A, B]′ composes the values bound to the variables A resp. B to a new pair.

The operators denoting concatenation and insertion - introduced below - will behave analogously.

3.8. Concatenation and its inverse operation

Assume we want to delete superfluous ′noop′ statements in statement lists. Then we need a rule

{ (L1 . [noop] . L2) => (L1 . L2) }

where the dot stands for concatenation and its inverse operation. The pattern partitions the sequence of statements into three subsequences such that the second one is ′[noop]′, and binds L1 to the first one and L2 to the third one. The expression concatenates L1 and L2 to a new sequence of statements.

By abstracting from ′noop′, we obtain an example for importing variables:

{ X => { L1 . [% X] . L2 => L1 . L2 } }

This is a function of second order. Given an argument x, it returns a function that removes an occurrence of x from a sequence.

Examples for patterns with concatenation:

′L1 . [while [C, B]] . L3′	matches lists of arbitrary length containing a ′while′ statement.
′[S, noop] . L′	matches lists whose second element is ′noop′.

The dot operator is a potential source of non-determinism:

L1 . [noop] . L2	matched against	[a1, noop, a2, noop]

where the subvalues ai are statements other than ′noop′, yields a set of two environments:

$\langle L1 \to [a1]; L2 \to [a2, noop]\rangle$ and $\langle L1 \to [a1, noop, a2]; L2 \to []\rangle$

Formally, the semantics of the dot operator is defined by a union over all possible partitions:

$$P\,(p\,.\,q)\;r\;v \;=\; \bigcup_{u\,.\,w\,=\,v} P\,(p)\;r\;u \oplus P\,(q)\;r\;w$$

This operator allows for selecting arbitrary subsequences, and is not contained in any of Hope, ML, and Miranda.

3.9. Tree fragments, insertion and horizontal cuts

The dot operator for patterns allows for partitioning values by vertical cuts into a left and a right hand side since it inverts concatenation. Now we want to introduce an operation - also not contained in Hope, ML, and Miranda - performing horizontal cuts to obtain an upper and a lower part. The upper part is not a complete tree; it contains a hole ′@′ denoting the place where the lower part was cut out.

In TrafoLa expressions, the operator ′ˆ′ denotes insertion of a value into the hole of a tree fragment:

ife [c, @, e]	ˆ t	= ife [c, t, e]
add @	ˆ [a, b]	= add [a, b]
ifs [c, @, []]	ˆ [s1, s2]	= ifs [c, [s1, s2], []]
[s1, @, s4]	ˆ [s2, s3]	= [s1, [s2, s3], s4]

The pattern operator ′ˆ′ inverts insertion as the dot operator inverts concatenation. When a pattern ′p ˆ q′ is matched against a value *v*, *v* is separated in all possible ways into two values *u* and *w* such that *u* contains exactly one hole and $v = u \,\hat{}\, w$ holds. Then *p* is matched against *u* and *q* against *w*:

$$P\,(p\,\hat{}\,q)\;r\;v \;=\; \bigcup_{u\,\hat{}\,w\,=\,v} P\,(p)\;r\;u \oplus P\,(q)\;r\;w$$

Examples:

Let v = mul [add [a, b], add [c, d]].

add [A, B] — does not match v

U ˆ add [A, B] — matches v in two ways:

<U → mul [@, add [c, d]]; A → a; B → b> and
<U → mul [add [a, b], @]; A → c; B → d>.

U ˆ mul [@, A] ˆ B — matched against v gives one solution only:

<U → @; A → add [c, d]; B → add [a, b]>

Both ′.′ and ′ˆ′ are associative such that no parentheses are needed in the last example.

3.10. ′Boolean′ pattern operators

The following pattern operators don’t have a direct correspondent in the world of expressions. They serve to extend or restrict the set of environments produced by pattern matches.

Intersection

The pattern ′p & q′ is used to specify that a value to be matched must satisfy both the requirements imposed by pattern *p* and by pattern *q*. If the pattern *p* is simply a variable - this is an important special case - we write ′V: q′ instead of ′V & q′ due to aesthetic reasons. Hope and ML contain only the special case (′&′ in Hope, ′as′ in ML) whereas Miranda contains nothing of this feature.

Example: S: (L1 . [W: while _] . L2) & *Stm**

matches any sequence of statements containing a ′while′ statement. The ′while′ statement is bound to W, its left context to L1, and its right context to L2, whereas the whole sequence is bound to S.

Formal definition: $P\ (p\ \&\ q)\ r\ v\ =\ P\ (p)\ r\ v \oplus P\ (q)\ r\ v$

Union

The pattern ′p | q′ matches all values matched by *p* or by *q* or by both *p* and *q*. The sets of environments produced by *p* and by *q* are simply joined together:

$$P\ (p \mid q)\ r\ v\ =\ P\ (p)\ r\ v\ \cup\ P\ (q)\ r\ v$$

Example: matching sums or products: A: (add [*Exp, Exp*] | mul [*Exp, Exp*])

The ′|′ operator is also a potential source of non-determinism. There are some problems with variables explained in the next section.

Complement

If *p* is a pattern matching some values, then ′!p′ is a pattern matching all but those values.

$$P\ (!p)\ r\ v\ =\ \text{if}\ \ P\ (p)\ r\ v = \varnothing\ \ \text{then}\ \ \{<>\}\ \ \text{else}\ \ \varnothing$$

Note that the pattern ′!p′ does not bind variables since there are no subvalues they could be bound to when ′!p′ matches i.e. *p* does not match. Thus, the pattern ′!!p′ is not equivalent to *p* - it matches the same values but does not bind variables. This is similar to Prolog's ′not′ predicate [4].

Examples

Pattern	Value	Result
add [A, B] & ! add [E, E]	add [1, 2]	$\{<A \rightarrow 1; B \rightarrow 2>\}$
add [A, B] & ! add [E, E]	add [1, 1]	$\varnothing$

Deterministic ′noop′ elimination rule:

{ (L1: (*Stm* & !noop)*) . [noop] . (L2: *Stm**) => L1 . L2 }

The sequence bound to L1 must not contain ′noop′ statements such that the rule eliminates the first occurrence of ′noop′. Later, we shall give examples for functions deleting all occurrences.

Assume we want to select a function call in an assignment:

assign [V, E] ^ F: *Fcall*

The additional constraint that the function call is not the whole expression may be expressed as assign [V, E & !@] ˆ F: *Fcall*

3.11. Binding variables in patterns

Let V(p) denote the set of variables that are eventually bound by *p*. It is defined recursively:

$$V(c) = V(\_) = V(T) = V(\%\,A) = V(!p) = \varnothing$$
$$V(A) = \{A\}$$
$$V(p\,q) = V(p\,.\,q) = V(p \hat{}\, q) = V(p\,\&\,q) = V(p \mid q) = V(p) \cup V(q)$$
$$V([p_1, \ldots, p_n]) = V(p_1) \cup \ldots \cup V(p_n) \qquad V([]) = \varnothing$$
$$V(p^*) = V(p+) = V(p)$$

Alternatives ′p | q′ must be considered a little bit closer. Their components independently try to match and bind only the variables contained in themselves.

For instance, the (strange) pattern ′A | B′ matches any value v and produces a set of two environments $\{<A \rightarrow v>, <B \rightarrow v>\}$. Such strange patterns are forbidden; both operands of an ′|′ operator must bind the same set of variables. A similar problem arises with ′p*′ when matching the empty sequence.

Adopting these restrictions, the sets of environments produced by a pattern will be uniform:

Theorem:

Let p be a normal pattern i.e. satisfying two restrictions:

1) For all subpatterns $q_1 \mid q_2$ of p, $V(q_1) = V(q_2)$ holds.
2) For all subpatterns q^* of p, $V(q) = \varnothing$ holds.

Then for all environments r_o, all values v and all environments r in $P(p)\, r_o\, v$,
the set of variables bound in r is just V (p)

The restriction to normality excludes some awkward transformations, and implies the equivalence of ′p & p′ with *p* for all patterns *p* due to the idempotence of ′⊕′ for uniform sets of environments.

3.12. Other operators

There are a few other pattern operators already integrated in TrafoLa (see [8]), but the generality of Huet's inverse application is not yet reached. Its full integration will be investigated soon.

4. Expressions and definitions

Besides patterns, TrafoLa contains two other basic syntactic sorts: expressions and definitions. Patterns serve to analyze values, whereas expressions are used to synthesize values. Definitions occur at the top level of TrafoLa and in **′let′** and **′letrec′** constructs and bind variables to values.

The partners of the PROSPECTRA project did not yet agree upon the handling of non-determinism introduced by the pattern operators ′.′, ′ˆ′, and ′|′. There are two variants of TrafoLa: a deterministic one (D-TrafoLa [8]) where each function returns exactly one value, and a non-deterministic one (N-TrafoLa [5]) where each function returns a set of values that is enumerated by backtracking as in Prolog [4]. Accordingly, the semantic function for expressions has different type:

D-TrafoLa: E: Expression $\rightarrow$ Env $\rightarrow$ Value
N-TrafoLa: E: Expression $\rightarrow$ Env $\rightarrow$ 2^{Value}

4.1. Unfunctional expressions

These expressions are quite straightforward and only enumerated here:

Constructors:	c
Variables:	A
Sequences:	$[e_1, \ldots, e_n]$
Application:	e e'
Concatenation:	e . e'
Insertion:	e ^ e'
Comparison:	e = e' resp. e != e'
′let′ expressions:	**let** d **in** e **end** resp. **letrec** d **in** e **end**
	where ′d′ is a definition (see below)

Application also comprises tree construction e.g. ′add [1, 2]′. The present evaluation strategy is call by value since the complex patterns - especially ′^′ - exclude lazy evaluation, and abstract syntax trees to be transformed are usually finite.

In N-TrafoLa, non-determinism is accumulated by these operations: if ′e′ denotes *n* values and ′e'′ denotes *m* values, then ′e . e'′ will denote up to $n \cdot m$ values.

4.2. Functional expressions

Functional expressions are abstraction by a pattern and superposition of functions. The abstraction has syntax ′{ p => e }′ where *p* is a pattern and *e* an expression. Its meaning depends on the handling of non-determinism:

N-TrafoLa: $E\,(\{p \Rightarrow e\})\,r \;=\; \lambda x. \bigcup_{r' \text{ in } P\,(p)\,r\,x} E\,(e)\,(r + r')$

If *p* does not match the argument *x*, the function returns $\emptyset$.

D-TrafoLa:

Here, abstraction contains an implicit ′cut′ operator as known from Prolog:

Let select: $2^{\text{Env}} \rightarrow \text{Env}$ be some mapping with select s $\in$ s if $s \neq \emptyset$

Then $E\,(\{p \Rightarrow e\})\,r \;=\; \lambda x.$ if $s = \emptyset$ then fail else $E\,(e)\,(r + \text{select } s)$

where $s \;=\; P\,(p)\,r\,x$

′fail′ is a special value indicating that the pattern *p* failed to match the argument value *x*. All expressions except the ′|′ construct below are assumed to be strict with respect to ′fail′ i.e. if one operand evaluates to ′fail′ then the whole expression will also.

The superposition of (partial) functions is denoted by ′|′. This is a common operator in functional languages, but in Hope, ML, and Miranda, it is allowed in the context $(p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n)$ only.

In D-TrafoLa, ′|′ is only applicable to functions, and the first operand dominates:

$$E\,(f \mid g)\,r \;=\; \lambda x.\ \text{if } E\,(f)\,r\,x = \text{fail then } E\,(g)\,r\,x \text{ else } E\,(f)\,r\,x$$

In N-TrafoLa, it may be applied to all TrafoLa expressions:

$$E\,(e \mid e')\,r \;=\; E\,(e)\,r \cup E\,(e')\,r$$

Both variants of TrafoLa might contain the facility to collect the set of solutions into one sequence. In D-TrafoLa, this is done by a second abstraction mechanism ′{ p => **all** e }′.

4.3. Definitions and top level declarations

Definitions have syntax $A_1 = e_1$ **and** ... **and** $A_n = e_n$
Instead of ′A = {p => e}′ , we may write ′A p = e′.

Top level declarations are written ′**dec** d′ if they shall not be recursive, and ′**rec** d′ otherwise.

4.4. Some syntactic sugar:

Original form:	Alternative syntax:
({p_1 => e_1} | .. | {p_n => e_n}) e	**case** e **of** {p_1 => e_1} | .. | {p_n => e_n} **end**
({true => e_1} | {false => e_2}) e	**if** e **then** e_1 **else** e_2 **end**
{X => g (f X)}	f ; g

4.5. Examples

All examples are given for D-TrafoLa.

Function to simplify ′if′ expressions:

dec Sim-if = { ife [true, T, _] => T } | { ife [false, _, E] => E }

Identity:	**dec**	I X	= X /* alternatively for **dec** I = {X => X} */
Totalization by identity:	**dec**	Total F	= (F | I)
Repetition:	**rec**	Repeat F	= (F; Repeat F) | I

′Total f v′ computes ′f v′. If it is defined, it is the result, otherwise the result is the original argument *v*. ′Repeat f′ repeatedly applies *f* until it is no longer possible.

Functionals for sequences

rec Map F = { [] => [] } | { [H] . T => [F H] . Map F T }
rec Extend F = { [] => [] } | { [H] . T => F H . Extend F T }

Note the difference: ′Map′ applies a function item by item to a list, whereas ′Extend′ performs a homomorphic extension of its argument function from items to lists.

With	**dec** Double X = [X, X]
we obtain	Double [a1, a2, a3] = [[a1, a2, a3], [a1, a2, a3]]
	Map Double [a1, a2, a3] = [[a1, a1], [a2, a2], [a3, a3]]
and	Extend Double [a1, a2, a3] = [a1, a1, a2, a2, a3, a3]

Other classical functionals:

rec Fold F X0 = { [] => X0 } | { [H] . T => (F H (Fold F X0 T)) }

where F is a binary function and X0 typically is its neutral element.

Example: Fold (*) 1 [1, 2, 3, 4, 5] = 120.

rec Filter P = { [] => [] } | { [H] . T => **if** P H **then** [H] **else** [] **end** . Filter P T }

′Filter′ removes all list items not satisfying the predicate P.

Three functions deleting all ′noop′ statements from sequences of statements:

Repeat	{S1 . [noop] . S2 => S1 . S2}
Extend	({noop => []} | {X => [X]})
Filter	{X => X != noop}

Flat insertion

Remember that insertion ′^′ treats its second argument as a unit:

[1, @, 4] ^ [2, 3] = [1, [2, 3], 4]

But it is not difficult to define a function ′flat-insert′ splicing its second argument into the hole of the first:

dec Flat-insert = { U ^ (L . [@] . R) => { X => U ^ (L . X . R) } }

This is a curried function with two arguments. The first argument is partitioned by the pattern ′U ^ (L . [@] . R)′ into an upper context U, a left context L, and a right context R of the hole. The second argument is bound to X.

Flat-insert (ifs [c, [s1, @], []])	[]	= ifs [c, [s1], []]	(U = ifs [c, @, []] L = [s1] R = [])
Flat-insert [1, @, 4]	[2, 3]	= [1, 2, 3, 4]	(U = @ L = [1] R = [4])
Flat-insert (add @)	[1, 2]	= fail	(pattern match fails)
Flat-insert [1, @, 3]	2	= error	(L . X . R = [1] . 2. [3] = error)

5. A complex transformation: removal of function calls

5.1. 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 first described in [11], and also presented in [12].

5.2. How to get a new identifier

We shall first define some functions relying on an implementation of identifiers as numbers. All functions except these will be independent from this specific implementation.

type *Id* = id *Num*

dec Leastid = id 0 /∗ *Id* ∗/

dec Max (id A) (id B) = **if** A < B **then** id B **else** id A **end** /∗ *Id* → *Id* → *Id* ∗/

dec Nextid (id A) = id (A + 1) /∗ *Id* → *Id* ∗/

′Leastid′ is the least identifier that may occur in a program. Naturally, we must be sure that this is guaranteed by the tools generating the actual program.

When given a program, we may construct a new identifier not occurring in it by looking for the maximal identifier in the program and then building a greater one by ′Nextid′.

```
dec Maxid    = { _ ^ X: Id => all X } ;                Fold Max Leastid
               /* computes list of all identifiers */ /* determines maximum of list */
dec Newid    = Maxid ; Nextid
```

5.3. Subtasks to be done

Our task is to transform function calls in a Pascal like programming language into procedure calls by adding a new parameter exporting the result. To do so, we must first achieve that all function calls occur as right hand side of assignments only. In a second step, these assignments are transformed into procedure calls, and function declarations into procedure declarations.

```
/* Func-to-proc: Program → Program */
dec Func-to-proc = Unnest-all ; Transform-all
```

5.4. Unnesting of function calls

The transformation ′Unnest-all′ will transform the program such that it subsequently contains only calls ′V := F (...)′.

```
/* Unnest-all: Program → Program */
dec Unnest-all = Repeat { P: Program => Unnest-1 (Newid P) P }
```

′Unnest-all′ repeatedly calls the function ′Unnest-1′ until it is no more defined. ′Unnest-1′ unnests just one function call and uses the new identifier ′Newid P′.

A type name such as ′*Exp*′ only matches complete values i.e. values without holes. We use ′*Exp@*′ to match upper fragments of expressions. The unnesting of function calls already having the desired form ′v := f (...)′, is avoided by the pattern ′assign [*Id*, *Exp*@ & !@]′.

```
/* Unnest-1: Id → Program → Program */
dec Unnest-1 NEW =
    { U ^ S: assign [Id, Exp@ & !@] ^ F: Fcall
          => Flat-insert U          [assign [NEW, F], S ^ NEW] } |
    /* v := ... f (...) ...  =>  NEW := f (...); v := ...NEW... */

    { U ^ S: pcall _ ^ F: Fcall
          => Flat-insert U          [assign [NEW, F], S ^ NEW] } |
    /* p (... f (...) ...)  =>  NEW := f (...); p (...NEW...) */

    { U ^ S: ifs [Exp@, Stm*, Stm*] ^ F: Fcall
          => Flat-insert U          [assign [NEW, F], S ^ NEW] } |
    /* if .. f (..) .. then .. else .. fi  =>  NEW := f (..); if ..NEW.. then .. else .. fi */

    { U ^ while [E: Exp@, SL: Stm*] ^ F: Fcall
          => Flat-insert U          [assign [NEW, F], while [E ^ NEW, SL . [assign [NEW, F]]]] }
    /* while .. f (..) .. do .. od
          => NEW := f (..); while ..NEW.. do .. ; NEW := f (..) od */
```

Note that we replace one statement by two; thus, we need the ′Flat-insert′ function to avoid the building of nested subsequences.

The function ′Unnest-1′ may be made a little bit more readable (hopefully) by introducing a bigger upper context ′BU′ and a name for the new assignment.

```
/* Unnest-1: Id → Program → Program */
dec Unnest-1 NEW =
{ (BU: Program@ & ! (_ ^ assign [Id, @])) ^ F: Fcall =>
        let AS = assign [NEW, F] in
                case BU of   { U ^ S: (assign _ | pcall _ | ifs [Exp@, Stm*, Stm*])
                                  => Flat-insert U [AS, S ^ NEW] } |
                             { U ^ while [E: Exp@, SL: Stm*]
                                  => Flat-insert U [AS, while [E ^ NEW, SL . [AS]]] }
                end
        end
}
```

5.5. The real transformation

We define the function ′Transform-all′ by repetition of a simpler function transforming functions with some specific name only:

```
/* Transform-all: Program → Program */
dec Transform-all = Repeat { P: (_ ^ function [FN, _, _, _])
                                  => Transform-1 FN (Newid P) P }
```

′Transform-1 FN PN′ transforms all functions named FN into procedures named PN. The function name is added as an additional formal parameter such that the assignment to the function name in the body will export the result of the call.

```
/* Transform-1: Id → Id → Program → Program */
dec Transform-1 FN PN =
    Repeat  ( { U ^ function [% FN, PL, D, B] => U ^ procedure [PN, PL . [FN], D, B] } |
              /* fun FN (...) => proc PN (... , FN) */
              { U ^ assign [V, fcall [% FN, EL]] => U ^ pcall [PN, EL . [V]] }
              /* V := FN (...) => PN (... , V) */
            )
```

Here, we don't need the function ′Flat-insert′ since we replace one by one: one function declaration by one procedure declaration, and one assignment with function call by one procedure call.

5.6. Optimization of the transformation

Each call to ′Newid′ always computes the maximal identifier in the program from scratch, but we know that it is always the previous new one. Thus, we store the actual maximal identifier at the root of the program tree.

```
/* Func-to-proc': Program → Program */
dec Func-to-proc' =     {P: Program => [Maxid P, P]} ;
                        Unnest-all' ; Transform-all' ;
                        {[_, P] => P}
```

We compute the maximum identifier once and store it at the root, then we transform, and omit it at the end.

```
/* Unnest-all', Transform-all': [Id, Program] → [Id, Program] */
dec Unnest-all' = Repeat { [MI, P] => let NI = Nextid MI in [NI, Unnest-1 NI P] end }
dec Transform-all' =
    Repeat   { [MI, P: (_ ^ function [FN, _, _, _])]
                   => let NI = Nextid MI in [NI, Transform-1 FN NI P] end }
```

The mappings 'Unnest-1' and 'Transform-1' may still be used, such that this optimization can be done with little effort.

6. Conclusion and future research

The ability to specify patterns partitioning trees by arbitrary vertical and horizontal cuts allows for the definition of powerful transformation rules. Usual features of functional languages may be used to combine individual rules to transformation programs. A polymorphic type discipline has to be developed including both high order functions and tree grammar like data types.

The most significant element of our language is the powerful patterns. Many algebraic properties concerning the semantic equivalence of patterns hold e.g. the associativity of the pattern operators '|', '&', '.', and '^' (see [7]). An abstract pattern matching machine has been designed having many degrees of freedom such that there will be a flexible trade-off between the amount of precomputation by analyzing the pattern and the efficiency of matching it against concrete values. A prototype implementation of D-TrafoLa is available in ML [13]; it was created by translating the semantic clauses of TrafoLa into ML and does not yet include data type definitions and occurrences of data type names in patterns.

Acknowledgement

I wish to thank B. Gersdorf, B. Krieg-Brückner, U. Möncke, and R. Wilhelm for many discussions of TrafoLa, and H. G. Oberhauser and the ESOP '88 referees for comments on earlier drafts of this paper.

References

[1] Bobrow, D. G., Raphael, B.: New Programming Languages for Artificial Intelligence Research, ACM Comp. Sur. 6, 153-174, (1974)

[2] Burstall, R., MacQueen, D., Sannella, D.: HOPE: An Experimental Applicative Language, Report CSR-62-80, Computer Science Dept., Edinburgh, (1980)

[3] CIP Language Group: The Munich Project CIP. Volume I: The wide spectrum language CIP-L, Springer, LNCS 183, (1985)

[4] Clocksin, F. W., Mellish, C. S.: Programming in Prolog, Springer (1981)

[5] Gersdorf, B.: A Functional Language for Term Manipulation, PROSPECTRA M.3.1.S1-SN-2.0, (1987)

[6] Heckmann, R.: A Proposal for the Syntactic Part of the PROSPECTRA Transformation Language, PROSPECTRA S.1.6-SN-6.0, (1987)

[7] Heckmann, R.: Semantics of Patterns, PROSPECTRA S.1.6-SN-8.0, (1987)

[8] Heckmann, R.: Syntax and Semantics of TrafoLa, PROSPECTRA S.1.6-SN-10.0, (1987)

[9] Huet, G., Lang., B.: Proving and Applying Program Transformations

Expressed with Second Order Patterns, Acta Inf. 11, 31-55, (1978)

[10] Krieg-Brückner, B.: Informal Specification of the PROSPECTRA System, PROSPECTRA M.1.1.S1-R-9.1, (1986)

[11] Krieg-Brückner, B.: Systematic Transformation of Interface Specifications, in: Partsch, H. (ed.): Program Specification and Transformation, Proc. IFIP TC2 Working Conf. (Tölz '86), North Holland, (1987)

[12] Krieg-Brückner, B.: Algebraic Formalisation of Program Development by Transformation, Springer, this volume, (1988)

[13] Milner, R.: The Standard ML Core Language, In: Polymorphism, Vol. II, Number 2, (Oct. 1985)

[14] Turner, D. A.: Miranda: a non-strict Functional Language with Polymorphic Types, Springer, LNCS 201, (1985)