PROSPECTRA
an ESPRIT Project

Notes on Trafola, IV
Revision of the Previous Work

Reinhold Heckmann

Universität des Saarlandes
6600 Saarbrücken
Bundesrepublik Deutschland

PROSPECTRA Study Note S.1.6-SN-9.2

1987-10-05

Distribution: public

ABSTRACT

Now our view of Trafola has been more consolidated, and we are able to revise some of the concepts introduced in the previous papers [1,2,3]. This paper contains the recent changes of the design of Trafola and the reasons for them. ′New′ Trafola is presented in another paper.

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. Introduction

In the previous papers [1], [2], and [3], we made a lot of proposals and discussed several alternatives. Now, our view of Trafola is more consolidated, and we are able to revise the earlier work.

After having briefly considered insertion, we shall propose a new syntax. Then we shall discuss the nature of sequences; there are two different possibilities having advantages as well as drawbacks. Finally, we shall discuss iteration (starred patterns), and mention some new concepts to be included in Trafola.

The syntax and semantics of revised Trafola will be presented in another paper.

2. Insertion

In [2], we proposed four kinds of insertion: many or zero values into many holes (Δ'), many values into many holes (Δ), many holes, but one value per hole (A), and many values into one hole (Λ).

Meanwhile, we decided in a meeting with university of Bremen that only the simple insertion Λ is needed. It is simplified further by the change of the nature of sequences as described in chapter 4 below. We shall now denote it by the symbol ^ .

3. New syntax

3.1. Drawbacks of the old syntax

Parentheses were mainly needed to resolve syntactic ambiguities, but they were also used to delimit sequences although this was not strongly required.

Example: (1, 2, 3) could also be written 1, 2, 3 , but the second form looks strange (at least I think so).

For empty sequences, the parentheses are strictly necessary; to write instead of () is quite impossible.

Nevertheless, we decided that parentheses may be dropped if no syntactic confusion arises:

((a, b, c), (d, e, f), (), g) = (a, b, c, d, e, f, g)
"comma means concatenation" (cited from [1], page 11)

This implies that we need stronger sequence delimiters <> to build lists of lists and to prevent concatenation (see section 5.7, page 22 in [1]). '<>' was introduced as special operator (grouping operator), with alternative syntax

$<v_1; ... ; v_n>$ stands for $(<> v_1, ... , <> v_n)$.

The consequence is that reader and language designer wonder whether < > is the empty case of $<v_1; ... ; v_n>$ i.e. an empty sequence, or of $<v_1, ... , v_n>$ i.e. a grouping operator without children.

"Comma means concatenation", this principle induces a lot of confusion:

if (not C: Exp, T: Stm, E: Stm) => if (C, E, T)
statlist (A: Stm*, noop, B: Stm*) => statlist (A, B)

These rules do what we would expect, but the right hand side of the second rule looks quite unnaturally since (A, B) does not denote a sequence of two elements, but the concatenation of the sequences A and B. (C, E, T) also denotes a concatenation, but the result is always a three elements list since C, E, and T are bound to sequences of length 1 due to the type restrictions.

Now assume we want to drop the types making the rules more concise:

if (not C, T, E) => if (C, E, T)
statlist (A, noop, B) => statlist (A, B)

At least one of the two rules does not what we intended. The key question is: does a free name match arbitrary values or only values of length 1? In [1], we used the second convention; then the 'if' rule is correct, but the 'statlist' rule works only with statement lists of length 3 where the second statement is 'noop'. In [3], we used the first convention i.e. a name matches any value; then the 'if' rule is not correct: applied to if (not c, t, e) , C matches c, but T may match () and E the pair (t, e) such that the result is if (c, t, e). This is clearly not what we intended.

Thus, the old syntax suffers from the overloading of parentheses and the use of comma to denote concatenation. Severe problems are patched by the artificial grouping operator and strange conventions that induced new problems.

3.2. Proposal of a new syntax

Parentheses () only serve to resolve syntactic ambiguities. Square brackets [] are used as sequence delimiters; the empty sequence is denoted by ′[]′.

Sequences of sequences are now allowed, e.g. [[1, 2], [3, 4]]. Thus, auxiliary operators to delimit subsequences such as ′statlist′ are no longer needed.

In old syntax, the symbol ′[]′ denoted the hole in upper contexts, now we shall use ′@′ for this purpose. Furthermore, we used square brackets in old syntax to denote the actual position during tree walks; now, we shall use a special marking operator ′$′.

Comma now only serves to separate the items in a sequence, and concatenation is denoted by dot ′.′. These symbols are used in patterns, expressions, and values.

Examples:

(1) if [not [C], T, E] => if [C, E, T]
(2) (A . [noop] . B) => (A . B)

A name matches any value (sequence of arbitrary length), but the pattern $[p_1, \ldots, p_n]$ matches only sequences $[v_1, \ldots, v_n]$ of length n and checks whether the patterns p_i match the values v_i.

Singleton lists must be carefully distinguished from their only element i.e. the brackets enclosing ′C′ in rule (1) and ′noop′ in rule (2) must not be dropped.

3.3. Comparison between old and new syntax

	old	new
Resolving syntactic ambiguities:	(...)	(...)
Empty sequence:	()	[]
First order sequence delimiters:	(...)	[...]
Higher order sequence delimiters:	<...>	[...]
Hole:	[]	@
Indicator of actual subtree:	[...]	$
Separator of items in sequences:	,	,
Concatenation:	,	.

4. The nature of sequences: soft versus hard

As mentioned in the previous chapter, there are now sequences of sequences, and a singleton list does not equal its only item. This decision and its consequences will now be discussed.

There are two principal ways to define sequences in computer science: there are ′hard′ sequences having something like a delimiting shell, and ′soft′ sequences without such a shell. Hard sequences occur in some programming languages e.g. Lisp, Hope, and ML, while soft sequences are used in the theory of free monoids over some alphabet. With hard sequences, sequences of sequences may be constructed, and there is a difference between sequential and primitive objects, whereas with soft sequences, there are no sequences of higher order, and a singleton sequence equals its only element. The following table gives an overview of the differences.

Soft solution	Hard solution
Singleton lists equal their only item:	Singleton lists are different from their only item:
$[x] = x$ for all x	$[x] \neq x$ for all x
No sequences of sequences	There are sequences of sequences
Primitives are also sequences:	Primitives and sequences are distinct:
$\mathrm{Prim} \subseteq \mathrm{Seq}$	$\mathrm{Prim} \cap \mathrm{Seq} = \varnothing$
[, ,] is only defined for primitives	[, ,] is total

′.′ is total	′.′ is only defined for sequences
Insertion into sequences: $[u_1, \ldots, u_n] \,\hat{}\, v$ =	
$[u_1, \ldots, u_{i-1}] \,.\, (u_i \,\hat{}\, v) \,.\, [u_{i+1}, \ldots, u_n]$	$[u_1, \ldots, u_{i-1}, u_i \,\hat{}\, v, u_{i+1}, \ldots, u_n]$
[1, @, 4] ^ [2, 3] = [1, 2, 3, 4]	[1, @, 4] ^ [2, 3] = [1, [2, 3], 4]

Why this difference in the definition of insertion? The definition makes the respective insertion as defined as possible by using the respective total operation: ′.′ for the soft solution, ′[, ,]′ for the hard solution.

The main difference is concerned with the two operations of sequence construction ′[,]′ and concatenation ′.′. Concatenation is total for soft sequences:
1 . [2, 3] = [1] . [2, 3] = [1, 2, 3];
this is impossible with hard sequences since $1 \neq [1]$.

In contrast to this, sequence construction is not defined on soft sequences, e.g. [[1, 2], [3 ,4]] is not defined.[1]

Advantages of the soft solution:

1) The user need not distinguish between x and [x].
2) Concatenation is total.

Advantages of the hard solution:

1) There are sequences of sequences.
2) Sequence construction is a total operation and may be inverted i.e. the items v_i can be reconstructed from the sequence $[v_1, \ldots, v_n]$.
3) ′Hard′ insertion is considerably easier to invert than ′soft′ insertion. Specially, the so-called ′degenerated partitions′ disappear.
4) There is a finer typing of values since primitive objects and sequences are distinguished.

To have sequences of sequences is a big advantage; assume for instance a function returning a pair [u, v] of values. This is possible even if u and v are again sequences, and the two values can be reconstructed from the pair by a pattern [U, V].

′Hard′ insertion allows only for a number of partitions of a given value that is linear in the size of the value, whereas ′soft′ insertion induces a quadratic number of partitions. There are no degenerated partitions with hard insertion since e.g. [1, @, 2] ^ [] = [1, [], 2] $\neq$ [1, 2].

Example:

Partitions of the value ′if [c, t, e]′ where ′c′, ′t′, and ′e′ are constants.

Soft insertion: if [c, t, e] =

@	^	if [c, t, e]
if @	^	[c, t, e]
if [@, e]	^	[c, t]
if [c, @]	^	[t, e]
if [@, t, e]	^	c
if [c, @, e]	^	t
if [c, t, @]	^	e
[@, if [c, t, e]]	^	[]
[if [c, t, e], @]	^	[]
if [@, c, t, e]	^	[]
if [c, @, t, e]	^	[]
if [c, t, @, e]	^	[]
if [c, t, e, @]	^	[]

Hard insertion: if [c, t, e] =

@	^	if [c, t, e]

[1] Even if it were defined to yield [1, 2, 3, 4], this would not cause a big increase of the usability of soft sequences.

if @	^	[c, t, e]
if [@, t, e]	^	c
if [c, @, e]	^	t
if [c, t, @]	^	e

Considering the advantages of the ′hard′ solution, we decide that Trafola will only contain hard sequences.

5. Iteration

Remember the definition of iteration in [3]: An iteration has syntax ′p {i..j}′ where $0 \le i \le j \le \infty$ and $i < \infty$. There are abbreviations p {i} for p {i..i}, p* for p {0..∞}, and p+ for p {1..∞}. Pattern p {i..j} means p {i} | ... | p {j}, and p {i} means p . ... . p i.e. iteration is repeated concatenation.

Now two questions arise:

1) Do we really need patterns such as p {17..35} ?

2) Should p {i} mean p . ... . p , or rather [p, ... , p] ?

We discuss the second question first. ′p . ... . p′ matches sequences concatenated from i subsequences matched by p. These subsequences have arbitrary length such that it is very difficult to check whether this pattern matches. The pattern ′[p, ... , p]′ matches sequences consisting of i items matched by p; this is relatively easy to check.

Now consider a starred pattern p*. With dot semantics, p* matches all sequences concatenated from an arbitrary number of subsequences of arbitrary length all matched by p. This is even more difficult to check than the matching of p {i}. But to check whether p* matches a sequence with ′[,]′ semantics, we need only check that p matches all components what is even simpler than the matching of p {i} since we need not count the components.

Thus, dot iteration is drastically more complex than ′[,]′ iteration, but more general since ′[,]′ iteration may be simulated by dot iteration:

$p\ \{i..j\}_{[,]} \simeq [p]\ \{i..j\}_{.}$

All our examples need only ′[,]′ iteration, and we decided to restrict ourselves to it due to the big difference in complexity.

Now we turn to the first question. With ′[,]′ semantics, p {i..j} is only a special case of p*; the difference is that the length of the matched value is restricted to the range [i..j]. Therefore and because patterns such as p {17..35} will probably never occur, we restrict ourselves to p* = p {0..∞} and p+ = p {1..∞}. All other iterations can be expressed by those; in fact, only one of them would be needed since

p* ≃ [] | p+ and p+ ≃ p* & ![] ≃ [p] . p* ≃ p* . [p]

but we think that both will often occur.

Examples how other iterated patterns may be expressed:

p {3}	≃ [p, p, p]
p {0..1}	≃ [] | [p]
p {2..4}	≃ [p, p] | [p, p, p] | [p, p, p, p]
p {4..∞}	≃ [p, p, p] . p+ ≃ [p, p, p, p] . p*
p {17..35}	≃ { A: p+ where Inrange [17, 35] (Length A) }

using the ′where′ pattern operator explained below with the Boolean expression ′Inrange [17, 35] (Length A)′ meaning ′$17 \le$ Length A ≤ 35′.

6. New concepts to be included

6.1. Operator patterns

The pattern ′op q′ where *op* is an operator name and *q* a pattern, will be generalized to ′p q′ where *p* and *q* are both patterns.

Example: O: (add | mul) [A, B] matches add [1, 2] producing

<O $\rightarrow$ add, A $\rightarrow$ 1, B $\rightarrow$ 2>.

6.2. Patterns restricted by expressions: the ′where′ operator

The pattern ′p where e′ - *p* being a pattern and *e* a (Boolean) expression - matches those values matched by *p* where *e* evaluates to ′true′ in the produced environment.

Examples:

(1) A: p+ where Inrange [17, 35] (Length A)

(2) A . B where Length A = Length B

(3) add [A, B] where A ≠ B ≃ add [A, B] & ! add [E, E]

′A . B′ matched against [1, 2] produces the set of environments
{<A $\rightarrow$ [], B $\rightarrow$ [1, 2]>, <A $\rightarrow$ [1], B $\rightarrow$ [2]>, <A $\rightarrow$ [1, 2], B $\rightarrow$ []>}
whereas ′A . B where Length A = Length B′ only produces
{<A $\rightarrow$ [1], B $\rightarrow$ [2]>}.

6.3. The definedness predicate

We propose a predicate ′Def′ meaning λf. λx. (f (x) is defined) such that ′Def F′ is the characteristic predicate of the domain of F. ′f (x) is defined′ means that no pattern match failures occur during the evaluation of f (x). If the evaluation of f (x) does not terminate, then ′Def f x′ will terminate neither.

Examples:

With F = {0 => 1}, ′Def F′ means ′λx. (x = 0)′.

Assume a function F is defined for some special trees. Now F shall be applied to a subtree of a bigger tree where we don't want to specify to which subtree; the only requirement is that F be defined on it.

Somewhere F = { U ^ X where Def F X => U ^ F X }

6.4. Importing values into the pattern match

Assume there is a function defined for pairs looking for an occurrence of the first component within the second:
F = { [X, q] => ... } where X is a variable and the pattern q contains an occurrence of X.
The matching of q is restricted since q must bind its X to the same value as the X of the first component does.

Now assume we want to ′curry′ this function:
F' = { X => { q => ... } }

F' is only equivalent to F if the match of q against the second argument is influenced by the fact that X is already bound to the first argument.

One solution is to fix that a variable in a pattern already bound to a value matches only this value, but this would cause strange effects by unintended influences from outer scopes. The solution that we shall adopt is to designate those variable that should import a value into a pattern.

Examples (the import symbol is ′%′):

F = { X => { X => X }} : no interference, the expression ′X′ refers to the inner pattern i.e. F is equivalent to { X => { Y => Y }}, and F 1 2 = 2.

G = { X => { %X => X }} :
the second argument must equal the first one i.e. the function is equivalent to
{ X => { Y where Y = X => X }},
and G 1 1 = 1, but G 1 2 = undef.

Remove = { X => { A . [%X] . B => A . B },
Remove 3 [1, 2, 3, 2, 3] = [1, 2, 2, 3]

6.5. Handling of non-determinism: simple and ′all′ abstraction

This section is to be considered a proposal, further research and discussions are needed in this area.

In most cases, we want deterministic functions while patterns are not deterministic. Thus, we define the application ′{p => e} v′ to mean that pattern *p* is matched against the value *v*, then one of the resulting environments is selected by some algorithm, and *e* is evaluated in this environment.

Sometimes, we want not only one result but a complete list of all possible results. Then, we may write ′{p => **all** e} v′ meaning that *e* is evaluated in all environments produced by the pattern match of *p* against *v*; the results are combined into a sequence.

The order of the items in this sequence and the algorithm to select an environment could be unknown to the user, or known, but unchangeable, or it could be alterable by the user. We don't want to decide this here.

Examples

Prefix = { A . _ => A }, Prefix [1, 2, 3] = [] or = [1, 2, 3] depending whether the selection models a shortest or longest match strategy.

Prefixes = { A . _ => **all** A },
Prefixes [1, 2, 3] = [[], [1], [1, 2], [1, 2, 3]]

References

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

[2] Heckmann, R.: The Objects of the Transformation Language and the Operations upon them, [S.1.6-SN-7.0]

[3] Heckmann, R.: Semantics of Patterns, [S.1.6-SN-8.0]