PROSPECTRA
an ESPRIT Project

Notes on Trafola, V
Syntax and Semantics of Trafola

Reinhold Heckmann

Universität des Saarlandes
6600 Saarbrücken
Bundesrepublik Deutschland

PROSPECTRA Study Note S.1.6-SN-10.0

1987-10-05

Distribution: public

ABSTRACT

In this paper, syntax and semantics of some part of Trafola are formally presented. It overrides the previous papers [1,2,3] containing a preliminary description of Trafola, by incorporating the changes discussed in [4].

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

This paper is a new updated version of [1,2,3], but also includes some new topics. The reasons of the updates were discussed in [4]; some new concepts were also introduced there.

We shall define the syntax and semantics of some part of Trafola; they will be presented more completely and more formally than ever before. We shall start with a short view at the names occurring in Trafola, then consider the semantic domain of values (cf. [2]) and environments (cf. [3]). Then syntax and semantics of patterns are defined (cf. [3]), and finally syntax and semantics of expressions and declarations (this is new).

In another paper, we shall give a lot of examples of functions written in Trafola including the big example of transforming function into procedure calls of [1].

In [1], we used tree grammars to specify syntactic sorts. They may also be considered data type definitions declaring special data types, or declarations of recursive patterns that are bound to names. Such syntactic sorts resp. data types resp. names bound to patterns could be used in other patterns e.g. ′Function-call′ in ′assign [V, F: Function-call]′. The semantics of such patterns bound to names is not yet quite clear, thus it will not be included here.

2. Names in Trafola

In Trafola, there are different classes of names:

1) Constant names
   a) Numbers: 0, 1, -1, 2, -2, ...
   b) Symbolic constants e.g. ′noop′, ′false′, ′true′
2) Operator names such as ′add′, ′sub′, ′while′
3) Variable names that may be bound to a value e.g. *I* and *X* in the declaration ′dec I = { X => X }′. *I* is bound to a function (the identity), and *X* is bound to the argument of this function when it is applied.
4) Key words such as ′if′, ′let′, etc.

We assume that these classes are disjoint and there is a method to decide which class a given occurrence of a name belongs to. This method is immaterial when defining the syntax and semantics abstractly, but for our examples, we must choose one. The decision of the class a name belongs to, might be done by considering the lexical structure of the name as in Prolog, or by considering the context (declarations) as in Pascal.

Since variables in patterns are not declared, we adopt the convention that the class be recognizable from the lexical structure:

1) Constant names
   a) Numbers consist only of digits, eventually preceded by ′-′.
   b) Symbolic constants start with ′/′ e.g. ′/noop′, ′/false′, ′/true′.
2) Operator names begin with a lower case letter: ′add′, ′sub′, ′while′.
3) Variable names begin with an upper case letter: ′I′, ′X′, ′Stm′.
4) There is only a finite number of keywords such that they need no special lexical form. Here, we shall print them in bold face: ′**let**′, ′**if**′.

Comments:

- The distinction between (2) and (3) is analogous to Prolog.
- We think that symbolic constants occur less often than operator and variable names such that it is not a big nuisance to precede them with ′/′.

3. The objects of the transformation language

The following chapter is intended to be a revision of [2]. Due to our new syntax and the restriction to ′^′, drastic simplifications are achieved, but on the other hand, there are new classes of values not included in [2].

Values are semantic objects. Variable names may be bound to values, values may be analyzed by patterns and synthesized by expressions. But note that there is no means to directly write down a value in Trafola; thus there is no need to give them a nice syntax. We do this nevertheless such that we are able to write down values in this paper, but the reader should carefully distinguish among the value ′[1, 2]′, the expression ′[1, 2]′ - that naturally evaluates to the value ′[1, 2]′ in every environment - and the pattern ′[1, 2]′ matching the value ′[1, 2]′ and nothing else. Similarly, he/she must distinguish among the dot operator ′.′ in patterns, expressions, and in the semantic domain of values. Perhaps, we should write them as ′pdot′, ′edot′, and ′dot′ respectively, but we hope it will always be clear which one is intended.

3.1. Definition of values

In contrast to [2], there are now some more classes of values. We shall enumerate them and give a standard notation for each class.

- The special value ′fail′ produced by failing pattern matches
- Constants /c where /c is a constant name
- Operators op where op is an operator name
- The hole @
- Trees op v where op is an operator and v an arbitrary value, but not ′fail′.
- Sequences $[v_1, \ldots, v_n]$ where $v_1, \ldots, v_n$ are values, but not ′fail′. A special case is the empty sequence [].
- Functions mapping values to values

We shall use lambda calculus to write functions, extended by an ′if-then-else′ construct and the fixed point combinator ′Y′.

To be precise, we had to distinguish between Booleans and Boolean values, semantic functions and functional values, etc., but we identify them for simplicity such that the domain of values includes numbers, Booleans, and functions.

The operations of tree and sequence construction are artificially extended to ′fail′. We define op fail = fail for each operator op, and $[v_1, \ldots, v_n]$ = fail if any of the v_i is ′fail′.

After having defined values, we shall now introduce some operations on them that we shall later need in the semantic clauses.

Some operations on values will not be total i.e. they are not defined for some arguments. If they are nevertheless applied to such arguments during an evaluation, an error results and the evaluation breaks off.

In the definition of such operations, this is indicated by the pseudo-value ′error′ that is not considered an ordinary value since it cannot be handled by the ′|′ combinator and the definedness predicate.

3.2. Class checks

We assume that a given value can uniquely be attached to the class it belongs to. This naturally leads to the respective predicates ′λv. v is a constant′ etc. We introduce names for the classes of values, namely Const, Op, {@}, Tree, Seq, Func, and {fail}, and shall write ′v $\in$ Const′ instead of ′v is a constant′. Sometimes, we shall use abbreviations like ′if v = $[v_1, \ldots, v_n]$′ whose meaning should be clear.

3.3. Equality

We need an operation = : Value X Value $\rightarrow$ Bool $\subseteq$ Value, but we know that the equality of functions is not decidable. We shall therefore use a restricted equality: whenever a functional value is created, it is associated with a unique label; and comparisons of functional values are done by comparing only this label.

Example:

dec I	= {X => X}	I is bound to (function 1, λv. v)
dec J	= I	J is bound to I's value i.e. (function 1, λv. v)
dec K	= {X => X}	K is bound to (function 2, λv. v)

Then I's and J's value are equal, but I's and K's are not.

These labels may also be used to decide the class of a given value. We shall omit them in our semantic clauses for simplicity.

An alternative solution would be to produce an error if functions are going to be compared.

The operation ′=′ shall be not strict relative to ′fail′ i.e. ′fail = @′ gives ′/false′. Sometimes, we shall need a strict equality

Eq (u, v) = if u or v = fail then fail else (u = v)

3.4. Application

Since there will be no syntactic difference between the application of a function and the construction of a tree in Trafola, we introduce here a general operation handling both:

apply: Value X Value $\rightarrow$ Value,
apply (u, v) =
  if u or v = fail then fail else
  if u in Op then u v else
  if u in Func then u v else
  error

Notice that the first occurrence of ′u v′ means the construction of a tree from the operator ′u′ and the value ′v′, whereas the second one means the application of the functional value ′u′ to the value ′v′.

There is a detail worth to be discussed: should really apply (u, fail) = fail hold, or rather apply (u, fail) = u fail for functions u? There is no big difference; with our definition, {p => **all** e} (F X) = fail holds if F fails, whereas the alternative definition would cause the result to be [].

3.5. Length and number of holes

Each sequence *s* has a length |s|. We do not define a length of other values. The number of holes in a value *v* is denoted by ′hole (v)′. In [2], we called these two numbers upper length *UL* and lower length *LL*, but these names were easy to change by mistake.

Formal recursive definition:

Definition: | |, hole: Value $\rightarrow$ $\mathbf{N}_o$

|fail| = fail | [v_1, ... , v_n] | = n, specially | [] | = 0
For other values v: |v| = error
hole (fail) = fail   hole (@) = 1   hole (op v) = hole (v)
hole ([]) = 0   hole ([v_1, ... , v_n]) = hole (v_1) + ... + hole (v_n)
For other values v: hole (v) = 0

|_| and ′hole′ may independently vary in $\mathbf{N}_o$ except that |v| = 0 implies hole (v) = 0.

3.6. Concatenation

In the set *Seq*, we introduce the operation of concatenation ′.′ with neutral element [].

Definition: . : Value X Value $\rightarrow$ Value,

fail . v = u . fail = fail
[v_1, ... , v_n] . [w_1, ... , w_m] = [v_1, ... , v_n, w_1, ... , w_m] for all n, m $\geq$ 0.
u . v = error for all other values u, v.

Note that concatenation is not defined for trees or ′@′, i.e. ′1.2′, ′1.[]′, or ′@.@′ are undefined.

In the following proposition about the properties of ′.′, we shall use the special equal signs ′=′ and ′≐′ of [2], page 10 or 29. ′u ≐ v′ means ′u = error or u = fail or u = v′. The formulae are implicitly quantified over values, such that we must check whether both sides of the equations are equally defined. This is for instance not true for the equation ′v . [] = v′.

Proposition 1

$u\ .\ (v\ .\ w) = (u\ .\ v)\ .\ w$

$v\ .\ [] = []\ .\ v \doteq v$

$u\ .\ v = u\ .\ v'$ and $u \neq fail$ implies $v = v'$

$u\ .\ v = u'\ .\ v$ and $v \neq fail$ implies $u = u'$

$u\ .\ v = []$ iff $u = v = []$

$|[]| = hole\ ([]) = 0$

$|u\ .\ v| = |u| + |v|$

$hole\ (u\ .\ v) \doteq hole\ (u) + hole\ (v)$

The set ′Seq′ is a monoid under concatenation, and |_| and ′hole′ are monoid homomorphisms from (Seq, .) to $(\mathbf{N}_o, +)$.

3.7. Insertion

We now define the ′simple′ insertion ′^′ of an arbitrary value into one hole.

Definition: ^: Value X Value → Value

if $u = fail$ or $v = fail$ then $u\ \hat{}\ v = fail$
if $hole\ (u) \neq 1$ then $u\ \hat{}\ v = error$, otherwise
$@\ \hat{}\ v = v$
$(op\ u)\ \hat{}\ v = op\ (u\ \hat{}\ v)$
$[u_1, \dots , u_n]\ \hat{}\ v = [u_1, \dots , u_{i-1}, u_i\ \hat{}\ v, u_{i+1}, \dots , u_n]$
where $hole\ (u_i) = 1$ and $hole\ (u_j) = 0$ for $j \neq i$.
(u_i is the component containing the hole of u).

Examples:

op @ ^ v = op v
add @ ^ [1, 2] = add [1, 2]
add [1, @] ^ 2 = add [1, 2]
[1, @, 2, 3] ^ [] = [1, [], 2, 3]
[1, @, 3, 4] ^ 2 = [1, 2, 3, 4]
[1, @, 3, 4] ^ [2] = [1, [2], 3, 4]
[1, @, 4, 5] ^ [2, 3] = [1, [2, 3], 4, 5]
add [@, @] ^ [1, 2] is undefined (too many holes)
add [1, 2] ^ [] is undefined (no hole in first operand)

It will turn out that we cannot partition w = [1, 2, 3, 4] into u and v such that w = u ^ v, u contains 1 and 4, and v contains 2 and 3. But this partition can be done by ′.′.

Proposition 2

(1) $(op\ u)\ \hat{}\ v = op\ (u\ \hat{}\ v)$
(2) If $u\ \hat{}\ v$ is defined and not ′fail′, and $u \neq @$ then
  (a) u in Tree iff $u\ \hat{}\ v$ in Tree
  (b) u in Seq iff $u\ \hat{}\ v$ in Seq
(3) $hole\ (u\ \hat{}\ v) \doteq hole\ (v)$
(4) $|u\ \hat{}\ v| \doteq$ if $u \neq @$ then $|u|$ else $|v|$
(5) $@\ \hat{}\ v = v$ $u\ \hat{}\ @ \doteq u$
(6) $u\ \hat{}\ v = []$ iff $u = @$ and $v = []$
(7) $u\ \hat{}\ v = @$ iff $u = v = @$
(8) Let u and u’ contain one hole such that u ^ ... and u’ ^ ... are defined.
  (a) $u\ \hat{}\ v = u'\ \hat{}\ v$ and $v \neq fail$ does not imply $u = u'$.
  (b) $u\ \hat{}\ v = u\ \hat{}\ v'$ and $u \neq fail$ implies $v = v'$.
(9) if $hole\ (w) = hole\ (w') = 0$ then $(w\ .\ u\ .\ w')\ \hat{}\ v \doteq w\ .\ (u\ \hat{}\ v)\ .\ w'$
  Right hand side defined, but left hand side undefined iff $u = @$ and v in Seq
(10) $u\ \hat{}\ (v\ \hat{}\ w) = (u\ \hat{}\ v)\ \hat{}\ w$

Proof:

The case where one of the involved values is ′fail′, is not included in the proofs. It is either impossible, or the formulae degenerate to ′fail ≐ fail′, ′fail = fail′, or ′fail ≐ something else′.

(1) directly from the definition

(2a) u in Tree and $u \,\hat{}\, v$ defined imply $u = \mathrm{op}\ u'$
Then $u \,\hat{}\, v = \mathrm{op}\ (u' \,\hat{}\, v)$ in Tree.

(2b) u in Seq means $u = [u_1, \ldots, u_n]$.
Then $u \,\hat{}\, v = [u_1, \ldots, u_{i-1}, u_i \,\hat{}\, v, u_{i+1}, \ldots, u_n]$ in Seq.

(2) The opposite directions are true since the definedness of $u \,\hat{}\, v$ implies that u is in *Tree*, *Seq*, or $\{@\}$. $u = @$ is excluded by the precondition, and *Tree* and *Seq* are disjoint.

(3) $\mathrm{hole}\ (u \,\hat{}\, v) \doteq \mathrm{hole}\ (v)$
Induction on u.
Case $u = @$: true since $@ \,\hat{}\, v = v$
Case $u = \mathrm{op}\ u'$: $\mathrm{hole}\ (u \,\hat{}\, v) = \mathrm{hole}\ (\mathrm{op}\ (u' \,\hat{}\, v)) = \mathrm{hole}\ (u' \,\hat{}\, v) = \mathrm{hole}\ (v)$
Case $u = [u_1, \ldots, u_n]$:
Choose the index i such that $\mathrm{hole}\ (u_i) = 1$ and $\mathrm{hole}\ (u_j) = 0$ for $j \neq i$.
Then $\mathrm{hole}\ (u \,\hat{}\, v) = \mathrm{hole}\ ([u_1, \ldots, u_{i-1}, u_i \,\hat{}\, v, u_{i+1}, \ldots, u_n])$
$= \mathrm{hole}\ (u_1) + \ldots + \mathrm{hole}\ (u_{i-1}) + \mathrm{hole}\ (u_i \,\hat{}\, v) + \mathrm{hole}\ (u_{i+1}) + \ldots + \mathrm{hole}\ (u_n)$
$= \mathrm{hole}\ (u_i \,\hat{}\, v) = \mathrm{hole}\ (v)$ by induction

(4) Let u and v be values such that $|u \,\hat{}\, v|$ is defined, i.e. $\mathrm{hole}\ (u) = 1$ and $u \,\hat{}\, v$ in Seq.
Case $u = @$: Then $u \,\hat{}\, v = v$, thus v in Seq, and $|u \,\hat{}\, v| = |v|$.
Case $u \neq @$: Then u in Seq by (2b), i.e. $|u|$ is defined.
$u = [u_1, \ldots, u_n]$ implies $u \,\hat{}\, v = [u_1, \ldots, u_{i-1}, u_i \,\hat{}\, v, u_{i+1}, \ldots, u_n]$
for some i i.e. $|u \,\hat{}\, v| = |u|$.

(5) $@ \,\hat{}\, v = v$ by definition; note that $@ \,\hat{}\, v$ is always defined.
$u \,\hat{}\, @ \doteq u$ Note that $u \,\hat{}\, @$ may not be defined.
The statement is proved by induction on u.
Case $u = @$: $u \,\hat{}\, @ = @ \,\hat{}\, @ = @ = u$
Case $u = \mathrm{op}\ u'$: $u \,\hat{}\, @ = \mathrm{op}\ (u' \,\hat{}\, @) = \mathrm{op}\ u' = u$
Case $u = [u_1, \ldots, u_n]$:
Choose the index i such that $\mathrm{hole}\ (u_i) = 1$ and $\mathrm{hole}\ (u_j) = 0$ for $j \neq i$.
Then $u \,\hat{}\, @ = [u_1, \ldots, u_{i-1}, u_i \,\hat{}\, @, u_{i+1}, \ldots, u_n]$
$= [u_1, \ldots, u_{i-1}, u_i, u_{i+1}, \ldots, u_n] = u$

(6) $u \,\hat{}\, v = []$ iff $u = @$ and $v = []$:
$@ \,\hat{}\, []$ is indeed $[]$.
For the opposite direction, assume $u \,\hat{}\, v = []$. If u were not @, u would be in Seq by (2b), and $|u| = |u \,\hat{}\, v| = 0$ by (4). Thus u would be [], and $u \,\hat{}\, v$ would not be defined. Therefore, $u = @$ holds, and $v = @ \,\hat{}\, v = u \,\hat{}\, v = []$.

(7) $u \,\hat{}\, v = @$ iff $u = v = @$.
$@ \,\hat{}\, @$ is indeed @.
For the opposite direction, assume $u \,\hat{}\, v = @$. If u were not @, u would be either in Tree or in Seq, and $u \,\hat{}\, v$ would, too, by (2). Thus u is @, and $v = @ \,\hat{}\, v = u \,\hat{}\, v = @$.

(8) Example: $[@, c] \,\hat{}\, c = [c, c] = [c, @] \,\hat{}\, c$
The positive statement is proved by induction on u.
Case $u = @$: $v = @ \,\hat{}\, v = @ \,\hat{}\, v' = v'$
Case $u = \mathrm{op}\ u'$:
$(\mathrm{op}\ u') \,\hat{}\, v = (\mathrm{op}\ u') \,\hat{}\, v'$ implies $\mathrm{op}\ (u' \,\hat{}\, v) = \mathrm{op}\ (u' \,\hat{}\, v')$,
thus $u' \,\hat{}\, v = u' \,\hat{}\, v'$, and hence $v = v'$ by induction.
Case $u = [u_1, \ldots, u_n]$: Then $u \,\hat{}\, v = [u_1, \ldots, u_{i-1}, u_i \,\hat{}\, v, u_{i+1}, \ldots, u_n]$,
$u \,\hat{}\, v'$ analogous with v replaced by v'.
It follows $u_i \,\hat{}\, v = u_i \,\hat{}\, v'$ and $v = v'$ by induction.

(9) $(w \,.\, u \,.\, w') \,\hat{}\, v$ is defined iff $\mathrm{hole}\ (w \,.\, u \,.\, w') = 1$ and w, u, w' in Seq.
$w \,.\, (u \,\hat{}\, v) \,.\, w'$ is defined iff $\mathrm{hole}\ (u) = 1$ and w, $u \,\hat{}\, v$, w' in Seq.
Since $\mathrm{hole}\ (w) = \mathrm{hole}\ (w') = 0$, $\mathrm{hole}\ (w \,.\, u \,.\, w') = \mathrm{hole}\ (u)$ holds;
u in Seq implies $u \,\hat{}\, v$ in Seq by (2b).
The right hand side is defined although the left hand side is undefined
iff $u \,\hat{}\, v$ in Seq, but u is not in Seq iff $u = @$ and v in Seq.
The equality itself is an immediate consequence of the definition of ′^′.

(10) $u \,\hat{}\, (v \,\hat{}\, w)$ is defined iff $\mathrm{hole}\ (u) = 1$ and $\mathrm{hole}\ (v) = 1$.

$(u \hat{} v) \hat{} w$ is defined iff hole $(u) = 1$ and hole $(u \hat{} v) = 1$.
This is the same by (3).
The equality is proved by induction on u.
Case $u = @$: $@ \hat{} (v \hat{} w) = v \hat{} w = (@ \hat{} v) \hat{} w$
Case $u = op\ u'$:
$(op\ u') \hat{} (v \hat{} w) = op\ (u' \hat{} (v \hat{} w)) = op\ ((u' \hat{} v) \hat{} w) = (u \hat{} v) \hat{} w$
Case $u = [u_1, ... , u_n]$:
Then $u \hat{} (v \hat{} w) = [u_1, ... , u_{i-1}, u_i \hat{} (v \hat{} w), u_{i+1}, ... , u_n]$
$= [u_1, ... , u_{i-1}, (u_i \hat{} v) \hat{} w, u_{i+1}, ... , u_n]$
$= (u \hat{} v) \hat{} w$ by two applications of the definition of ′^′.

3.8. Number of partitions

Finally, we investigate how many partitions into two values a given value has. The result is much less complex than in [2] since $u \hat{} v$ is defined whenever hole $(u) = 1$ whereas the operation Λ in [2] had an additional condition for v to be defined, and we can only extract subtrees or complete lists of subvalues, not sublists.

Definition:

Let □ be a binary operation on values, and let w be a value. Then we denote the number of pairs u, v such that u □ v equals w, by $n_{\square}$ (w).

We shall not consider the special value ′fail′ since it will never be partitioned.

Concatenation

$n_{.}$ (w) = if w in Seq then $|w| + 1$ else 0
since $[w_1, ... , w_n] = [w_1, ... , w_i] . [w_{i+1}, ... , w_n]$ for $0 \le i \le n$

Values not being sequences cannot be partitioned by ′.′.

Insertion

$n_{\hat{}}$ ([])	= 1:	[]	=	$@ \hat{} []$
$n_{\hat{}}$ (@)	= 1:	@	=	$@ \hat{} @$
$n_{\hat{}}$ (/c)	= 1:	/c	=	$@ \hat{} /c$

$n_{\hat{}}$ (op w) = $1 + n_{\hat{}}$ (w)
op w = $(op\ u) \hat{} v$ = $@ \hat{} (op\ w)$
The first partition is possible whenever $w = u \hat{} v$ ($n_{\hat{}}$ (w) partitions), and the second one is always defined (1 partition).

Case $w = [w_1, ... , w_n]$:
If $w = u \hat{} v$, then u must contain exactly one hole i.e. all holes occurring in w must be extracted into v.

Case w contains holes.
Let k be the minimal index and l the maximal index such that hole $(w_k) > 0$ resp. hole $(w_l) > 0$.

If $k < l$, there is exactly one partition: $w = @ \hat{} w$.

If $k = l$, then we may extract a part of w_k or the whole value w.
(1) $w = [w_1, ... , w_{k-1}, u, w_{k+1}, ... , w_n] \hat{} v$ where $u \hat{} v = w_k$.
(2) $w = @ \hat{} w$
Thus we obtain $n_{\hat{}}$ (w) = $1 + n_{\hat{}}$ (w_k)

Case hole (w) = 0 i.e. w does not contain holes.
(1) $w = [w_1, ... , w_{i-1}, u, w_{i+1}, ... , w_n] \hat{} v$ where $u \hat{} v = w_i$, $1 \le i \le n$.
(2) $w = @ \hat{} w$

Number of partitions: $1 + n_{\hat{}}(w_1) + ... + n_{\hat{}}(w_n)$

Summary:
$n_{\hat{}}$ $[w_1, ... , w_n]$ =
1 — if w_k (w_l) is the first (last) component containing a hole and $k < l$
$1 + n_{\hat{}}$ (w_k) — if w_k is the only component containing a hole

$1 + n^{\wedge}(w_1) + ... + n^{\wedge}(w_n)$ if no component contains a hole

The formulae for sequences even hold if n = 0:
Last formula with n = 0:
$n^{\wedge}([]) = 1 + 0 = 1$

Examples:
$n^{\wedge}$ (add [/a, @]) = 1 + $n^{\wedge}$ ([/a, @]) = 1 + 1 + $n^{\wedge}$ (@) = 3
The three partitions are @ ^ add [/a, @], add @ ^ [/a, @], and add [/a, @] ^ @.
$n^{\wedge}$ (add [/a, /b]) = 1 + $n^{\wedge}$ ([/a, /b]) = 1 + 1 + $n^{\wedge}$ (/a) + $n^{\wedge}$ (/b) = 4
The four partitions are:
@ ^ add [/a, /b], add @ ^ [/a, /b], add [@, /b] ^ /a, add [/a, @] ^ /b.
$n^{\wedge}$ (if [eq [/a, /b], add [/a, @], sub [/b, @]]) = 1 + 1 = 2
$n^{\wedge}$ (if [eq [/a, @], add [/a, /b], sub [/a, /b]]) = 1 + 1 + $n^{\wedge}$ (eq [/a, @]) = 2 + 3 = 5
$n^{\wedge}$ (if [eq [/a, /b], add [/a, /b], sub [/a, /b]])
= 1 + 1 + $n^{\wedge}$ (eq [/a, /b]) + $n^{\wedge}$ (add [/a, /b]) + $n^{\wedge}$ (sub [/a, /b]) =
= 2 + 4 + 4 + 4 = 14
The fourteen partitions are illustrated in the following picture:

4. Environments and sets of environments

The semantic domains

$$\text{Env} = \text{Variable} \rightarrow (\text{Value} \cup \{\text{unbound}\})$$
$$\text{Sem} = 2^{\text{Env}}$$

and their operations have already been introduced in [3].

Standard notation for environments:

$\langle A_1 \rightarrow v_1, ... , A_n \rightarrow v_n \rangle =$
$\lambda X.$ if $X = A_1$ then v_1 else ... if $X = A_n$ then v_n else unbound

Standard notation for sets of environments: $\{r_1, ... , r_n\}$

A special environment is the empty one $\langle\rangle = \lambda X.$ unbound.

Now, we shall enumerate and update the operations in these domains:

+: Env X Env → Env,
$a + b = \lambda X.$ if b(X) = unbound then a(X) else b(X)
a and b are consistent with each other, iff
for all variables X, a(X) = unbound or b(X) = unbound or a(X) = b(X)
E: Sem, E = {< >}
$\cup$: Sem X Sem → Sem, $s \cup t = \{r \mid r \text{ in } s \text{ or } r \text{ in } t\}$
$\oplus$: Sem X Sem → Sem, $s \oplus t = \{a + b \mid a \text{ in } s, b \text{ in } t, a \text{ and } b \text{ are consistent}\}$
select: Sem - $\{\emptyset\}$ → Env, with select $s \in s$

The notion of consistence and the mapping 'select' are new. The superposition of environments '+' has been redefined, now the second argument overrides the first one if they are inconsistent with each other.

Union ′∪′ and superposition ′⊕′ of sets of environments are equal to the respective operations in [3], although the definition of ′⊕′ was altered. The ′select′ operation is only loosely specified: it selects an environment from a non-empty set.

The algebraic properties of ′∪′ and ′⊕′ are listed in [3], they are not repeated here.

5. Syntax and semantics of patterns

5.1. Introduction

This section is a revision of [3]. Among others, there are now Boolean expressions to restrict patterns, and the possibility to import a value bound to a variable (see [4]). Thus, the semantic function on patterns has type

P: Pattern → Env → Value → Sem

with an additional argument ′Env′ - the importing environment. It is distributed to all subpatterns without any change and used by importing variables ′% A′ and restricting expressions ′**where** e′.

5.2. Atomic patterns

The set of atomic patterns is influenced by the feature of binding patterns to variables. If this is not possible (as actually in this description), we need a lot of atomic patterns for different purposes. If there are variables bound to patterns, we need only some basic atomic patterns since the other ones might be bound to variable names by the user or by the system, e.g.

```
Tree   = _ _
Seq    = [_]*
Val0   = { A where Hole A = 0 }
Tree0  = Tree & Val0     (or should we write:  % Tree & % Val0 ?)
Fun    = /* predefined by system */
```

Thus, we shall propose only some basic atomic patterns. Their semantic function associates them with a predicate telling whether they match:

A: At → Value → Bool

@		At (@) v	= (v = @)
/c	(constant name)	At (/c) v	= (v = /c)
op	(operator name)	At (op) v	= (v = op)
_	(wild card)	At (_) v	= (v ≠ fail)

5.3. Syntax of patterns

Each line is an alternative, compare with [3].

(1)	at	atomic patterns
(2)	% N	variable importing some value
(3)	N	variable to be bound, matches any value
(4)	p ∣ q	′or′ operator (alternative)
(5)	p & q	′and′ operator
	N : p	an alternative syntax for ′N & p′
(6)	! p	′not′ operator
(7)	{ p }	′unbind′ operator
(8)	p q	generalization of ′op q′ in [3]
(9)	[p_1, ... , p_n]	sequence operator (a new pattern operator)
(10)	p*	iteration from 0 to ∞
(11)	p+	iteration from 1 to ∞
(12)	p . q	inverse of concatenation
(13)	p ˆ q	inverse of insertion

(14) p **where** e new operator: e is a Boolean expression

A pattern $[p_1, \ldots, p_n]$ only matches values of length n whose components are matched by the subpatterns p_i.

The construct ′p q′ is a generalization of the former construct ′op q′. *p* should be an operator pattern i.e. a pattern matching some operators.

The **where** operator allows for the inclusion of predicates into patterns; thus, it is a window from the world of patterns into the world of expressions and functions.

The syntax introduced above is not yet concrete; there are ambiguities to be solved by parentheses ′()′ and priorities. In our examples, we shall use some specific order of priorities that is somewhat arbitrary, but globally seems to be useful. In the following list, the operators are ordered by decreasing priority. Under each operator, its associativity is given: ′L′ (′R′) means the operator associates to the left (right) whereas ′A′ means that the operator is semantically associative.

′!′	′*′, ′+′	′:′	′ ′	′.′	′^′	′&′	′where′	′|′	′,′
R	L	R	R	A	A	A	L	A	A

′ ′ is the operator in the construct ′p q′. Comma, the operator with least priority, may only occur inside brackets ′[′ ′]′. Brackets and braces ′{′ ′}′ don’t occur in the list above since they need no priority.

Examples:

!p*	= (!p)*	
op p*	= op (p*)	
op (A . op B . c)	= op (A . (op B) . C)	(a nonsense pattern)
p & q | r	= (p & q) | r	
[p, q ^ r, s]	= [p, (q ^ r), s]	
U ^ S: p | q **where** e ^ X	= (U ^ (S: p)) | (q **where** (e ^ X))	
p | A: q | r	= p | (A: q) | r	

5.4. Denotational semantics of patterns

We shall describe patterns non-deterministically, i.e. a pattern applied to a value yields a set of environments. If it does not match the value, it yields the empty set.

Semantics of Patterns P: Pattern $\rightarrow$ Env $\rightarrow$ Value $\rightarrow$ Sem

(0) P (p) r fail = $\varnothing$ for all patterns p and environments r

(1) P (at) r v = if At (at) v then E else $\varnothing$

(2) P (% A) r v = if r(A) = unbound then error else if r(A) = v then E else $\varnothing$

(3) P (A) r v = {<A $\rightarrow$ v>}

(4) P (p | q) r v = P (p) r v $\cup$ P (q) r v

(5) P (p & q) r v = P (p) r v $\oplus$ P (q) r v

(6) P (!p) r v = if (P (p) r v = $\varnothing$) then E else $\varnothing$

(7) P ({p}) r v = if (P (p) r v $\neq \varnothing$) then E else $\varnothing$

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

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

(10) P (p*) r v = if v = $[v_1, \ldots, v_n]$
then P (p) r $v_1 \oplus \ldots \oplus$ P (p) r v_n
else $\varnothing$

(11) P (p+) r v = if v = $[v_1, \ldots, v_n]$ and v $\neq$ []
then P (p) r $v_1 \oplus \ldots \oplus$ P (p) r v_n
else $\varnothing$

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

(13) P (p ^ q) r v = $\bigcup_{u,\, w \text{ with } u\,\hat{}\,w = v}$ P (p) r u $\oplus$ P (q) r w

(14) P (p **where** e) r v = { r’ in P (p) r v | E (e) (r + r’) = /true }

Remark to the ′where′ construct:

All environments produced by *p* are filtered through the Boolean expression *e*. To check whether an environment *r'* produced by *p* is valid, *e* is evaluated in the superposed environment *r + r'* where *r* is the imported environment. ′E′ is the semantic function of Trafola expressions with functionality E: Expression → Env → Value. It will be specified below.

Remark to (9) and (10):

The empty case of $s_1 \oplus ... \oplus s_n$ for sets of environments s_i is E, the neutral element of ′⊕′. Thus, we obtain:

P ([]) r v = if v = [] then E else ∅
P (p*) r [] = E

From [3], we know that all sets P (p) r v are finite. The proof is principally still valid, but much simpler due to the restriction of iteration.

5.5. Examples

Let /c be a constant, u, v, w pairwise different values, ′op′ an operator name, and A, B, C variable names.

Pattern	Value	Result
A . B	/c	{}
A . B	[]	{<A → [], B → []>}
A . B	[u, v, w]	{<A → [], B → [u,v,w]>, <A → [u], B → [v,w]>, <A → [u,v], B → [w]>, <A → [u,v,w], B → []>}
A . /c . B	[/c, v, w]	{}
A . [/c] . B	[/c, v, w]	{<A → [], B → [v,w]>}
[A, /c, B]	[/c, v, w]	{}
[A, _, B]	[/c, v, w]	{<A → /c, B → w>}
A . A . B	[u,v,u,v,u]	{<A → [], B → [u,v,u,v,u]>, <A → [u,v], B → [u]>}
(A: ! []) . A . B	[u,v,u,v,u]	{<A → [u, v], B → [u]>}
A ˆ op [B, C]	op [u, op [v, w]]	{<A → @, B → u, C → op [v, w]>, <A → op [u, @], B → v, C → w>}

The pattern ′A . /c . B′ never matches, since a constant value cannot be concatenated.

op [A, B] & ! op [E, E]   or equivalent   op [A, B] where A != B
matched against op [u, v] yields {<A → u, B → v>}
matched against op [u, u] yields ∅.

An importing variable ′% A′ always refers to the importing environment i.e. to the context of the whole pattern, not to any occurrence of A within the pattern:

P (A: [%A, B]) <A → 1> [1, 2] = {<A → [1, 2], B → 2>}

Variables occurring in the expression *e* of a subpattern ′p **where** e′ of a bigger pattern *q* first refer to occurrences in *p*, then to the context of *q*, but never to occurrences in *q* without *p*.

′[A, B where A = B]′ is equivalent to ′[A, B: %A]′; it compares the imported value of *A* with the value just bound to *B*, whereas the occurrence of *A* in the first component of the pattern does not influence the evaluation of the expression:

P ([A, B where A = B]) <A → 2> [1, 2] = {<A → 1, B → 2>}
P ([A, B where A = B]) <A → 2> [1, 1] = {}
P ([A, B where A = B]) <> v gives an ′unbound′ error.

′[A, B] where A = B′ is equivalent to ′[A, B: A]′ and ′[A: B, B]′; it compares the two components of the pair with each other.

P ([A, B] where A = B) < > [1, 2] = {}
P ([A, B] where A = B) < > [1, 1] = {<A → 1, B → 1>}

5.6. Name sets and normality

Now we shall briefly remember some notions associated with patterns in [3]. Each pattern is associated with a set *N(p)* of names that **may** be bound by *p*, and the set *n(p)* of names that are guaranteed to be bound.

Definition: Name sets N, n: Pat $\to 2^{\text{Name}}$

(1)	N (at)	$= \emptyset$	n (at)	$= \emptyset$
(2)	N (% A)	$= \emptyset$	n (% A)	$= \emptyset$
(3)	N (A)	$= \{A\}$	n (A)	$= \{A\}$
(4)	N (p ∣ q)	$= N(p) \cup N(q)$	n (p ∣ q)	$= n(p) \cap n(q)$
(5)	N (p & q)	$= N(p) \cup N(q)$	n (p & q)	$= n(p) \cup n(q)$
(6)	N (!p)	$= \emptyset$	n (!p)	$= \emptyset$
(7)	N ({p})	$= \emptyset$	n ({p})	$= \emptyset$
(8)	N (p q)	$= N(p) \cup N(q)$	n (p q)	$= n(p) \cup n(q)$
(9)	N ([$p_1,...,p_n$])	$= N(p_1) \cup ... \cup N(p_n)$	n ([$p_1,...,p_n$])	$= n(p_1) \cup ... \cup n(p_n)$
(10)	N (p*)	$= N(p)$	n (p*)	$= \emptyset$
(11)	N (p+)	$= N(p)$	n (p+)	$= n(p)$
(12)	N (p . q)	$= N(p) \cup N(q)$	n (p . q)	$= n(p) \cup n(q)$
(13)	N (p ˆ q)	$= N(p) \cup N(q)$	n (p ˆ q)	$= n(p) \cup n(q)$
(14)	N (p **where** e)	$= N(p)$	n (p **where** e)	$= n(p)$

The clauses for ′N′ and ′n′ are analogous except (4) and (10). An alternative ′p ∣ q′ may bind all names occurring in *p* or *q*, but only the names occurring in both will be surely bound. The pattern ′p*′ when matched against ′[]′, always returns E i.e. there are no names bound by it.

A pattern p is called normal iff n (p) = N (p). From [3], we know that normality is sufficient for

$P(p)\, r\, v \oplus P(p)\, r\, v = P(p)\, r\, v$ for all environments r and values v.

5.7. The language described by a pattern

The language described by a pattern is the set of values matched by the pattern. It generally depends on the imported environment:

Relative language: RL (p) r = $\{v \text{ in Value} \mid P(p)\, r\, v \neq \emptyset\}$

Let ′abs p′ be the pattern resulting from *p* by changing all ′% A′ subpatterns into ′_′ and by removing all ′**where** e′ restrictions. Then ′abs p′ is independent from any imported environment, and we may define

Absolute language: AL (p) = RL (abs p) <>

Let ′free p′ be the pattern resulting from ′abs p′ by changing all variable names into ′_′. Then ′free p′ does not contain any variables.

Free language: FL (p) = AL (free p) = RL (free p) <>

Proposition:

For all patterns p and environments r, RL (p) r $\subseteq$ AL (p) $\subseteq$ FL (p) holds.

Examples:

RL ([A, A: %B]) $<B \to 1>$ = {[1, 1]}
AL ([A, A: %B]) = RL ([A, A: _]) <> = $\{[v, v] \mid v \in \text{Value}\}$
FL ([A, A: %B]) = AL ([_, _: _]) = $\{[v, w] \mid v, w \in \text{Value}\}$

Note that two equivalent patterns may have different absolute languages:

AL ([A, B: A]) = $\{[v, v] \mid v \in \text{Value}\}$
AL ([A, B] where A = B) = AL ([A, B]) = $\{[v, w] \mid v, w \in \text{Value}\}$

Some notions about patterns:
p is called degenerated (with respect to r) iff RL (p) r = $\emptyset$,
and absolutely degenerated iff AL (p) = $\emptyset$.
p is a tree pattern if AL (p) $\subseteq$ Tree,
a sequential pattern if AL (p) $\subseteq$ Seq,

and an operator pattern if $AL\ (p) \subseteq Op$.
Two new notions are the minimum and maximum length of a pattern:
$\mathrm{minlen}\ (p) = \min \{|v| \mid v \text{ in } AL\ (p) \cap Seq\}$
$\mathrm{maxlen}\ (p) = \max \{|v| \mid v \text{ in } AL\ (p) \cap Seq\}$
Naturally, ′minlen′ and ′maxlen′ are not defined if the pattern does not match any sequence. maxlen (p) may be ∞.
p has constant length iff minlen (p) = maxlen (p) (even if both are undefined)

5.8. Equivalence of patterns

Two patterns p and q are equivalent iff P (p) r v = P (q) r v for all environments r and values v. Most properties stated in [3] are still valid. We shall list them; the properties that don't generally hold will be provided with conditions for their validity.

Congruence properties

$p \simeq p$ $\quad$ $p \simeq q$ iff $q \simeq p$
$p \simeq q$ and $q \simeq r$ implies $p \simeq r$
If *p'* results from *p* by replacing some subpattern *q*
by an equivalent subpattern *q'*, then *p* and *p'* are equivalent, too.

Properties of | and &:

$p \mid q \simeq q \mid p$ $\quad$ $p \mid (q \mid r) \simeq (p \mid q) \mid r$ $\quad$ $p \mid p \simeq p$
$p \,\&\, q \simeq q \,\&\, p$ $\quad$ $p \,\&\, (q \,\&\, r) \simeq (p \,\&\, q) \,\&\, r$ $\quad$ $p \,\&\, \_ \simeq p$
$p \,\&\, (q \mid r) \simeq (p \,\&\, q) \mid (p \,\&\, r)$
$p \mid \_ \simeq \_$ $\quad$ if $N(p) = \emptyset$
$p \,\&\, p \simeq p$ $\quad$ if p is normal i.e. $n(p) = N(p)$
$p \,\&\, (p \mid q) \simeq p \mid (p \,\&\, q)$ $\quad$ if p is normal
$p \mid (p \,\&\, q) \simeq p$ $\quad$ if $N(q) \subseteq n(p)$
$p \,\&\, (p \mid q) \simeq p$ $\quad$ if $N(q) \subseteq n(p) = N(p)$
$p \mid (q \,\&\, r) \simeq (p \mid q) \,\&\, (p \mid r)$ $\quad$ if $N(q), N(r) \subseteq n(p) = N(p)$

Properties of name binding:

$A{:}\ (B{:}\ p) \simeq B{:}\ (A{:}\ p)$ $\quad$ $A{:}\ (A{:}\ p) \simeq A{:}\ p$
$A{:}\ (p \,\&\, q) \simeq (A{:}\ p) \,\&\, q \simeq p \,\&\, (A{:}\ q)$
$A{:}\ (p \mid q) \simeq (A{:}\ p) \mid (A{:}\ q)$

Properties of ! and {}:

$\{\{p\}\} \simeq \{p\}$ $\quad$ $\{!p\} \simeq\ !p$ $\quad$ $!\{p\} \simeq\ !p$ $\quad$ $!!p \simeq \{p\}$
$\{p \mid q\} \simeq \{p\} \mid \{q\}$ $\quad$ $!\ (p \mid q) \simeq\ !p \,\&\, !q$
$\{p\} \simeq p$ $\quad$ if $N(p) = \emptyset$
$\{p \,\square\, q\} \simeq \{p\} \,\square\, \{q\}$ $\quad$ if $N(p) \cap N(q) = \emptyset$ for $\square$ = ′&′, ′.′, ′^′, ′ ′
$\{[p]\} \simeq [\{p\}]$
$\{[p_1, \dots, p_n]\} \simeq [\{p_1\}, \dots, \{p_n\}]$ $\quad$ if $N(p_i) \cap N(p_j) = \emptyset$ for $i \neq j$
$!\ (p \,\&\, q) \simeq\ !p \mid !q$ $\quad$ if $N(p) \cap N(q) = \emptyset$

Properties of operators:
′op′ and ′oq′ denote patterns, not operator names in the sequel.

$op\ (p \mid q) \simeq (op\ p) \mid (op\ q)$	
$(op \mid oq)\ p \simeq (op\ p) \mid (oq\ p)$	
$op\ (p \,\&\, q) \simeq (op\ p) \,\&\, (op\ q)$	if op is normal
$(op \,\&\, oq)\ p \simeq (op\ p) \,\&\, (oq\ p)$	if p is normal
$\{op\ p\} \simeq \{op\}\ \{p\}$	if N(op) and N(p) are disjoint

Sequences

$$[p_1, \dots, p_n] \,.\, [q_1, \dots, q_m] \simeq [p_1, \dots, p_n, q_1, \dots, q_m]$$

$$[p_1, ..., p_n, p \mid q, q_1, ..., q_m] \simeq [p_1, ..., p_n, p, q_1, ..., q_m] \mid [p_1, ..., p_n, q, q_1, ..., q_m]$$

$$[p_1, ..., p_n, p \,\&\, q, q_1, ..., q_m] \simeq [p_1, ..., p_n, p, q_1, ..., q_m] \,\&\, [p_1, ..., p_n, q, q_1, ..., q_m]$$

if all p_i and all q_j are normal

Concatenation

$p \,.\, (q \,.\, r) \simeq (p \,.\, q) \,.\, r$	
$p \,.\, (q \mid r) \simeq (p \,.\, q) \mid (p \,.\, r)$	$(p \mid q) \,.\, r \simeq (p \,.\, r) \mid (q \,.\, r)$
$p \,.\, [] \simeq [] \,.\, p \simeq p \,\&\, Seq$	where 'Seq' matches only sequences
$p \,.\, q \,.\, [] \simeq p \,.\, [] \,.\, q \simeq [] \,.\, p \,.\, q \simeq p \,.\, q$	
$p \,.\, [] \simeq [] \,.\, p \simeq p$	if p is a sequential pattern
$p \,.\, (q \,\&\, r) \simeq (p \,.\, q) \,\&\, (p \,.\, r)$	if p is normal and has constant length
$(p \,\&\, q) \,.\, r \simeq (p \,.\, r) \,\&\, (q \,.\, r)$	if r is normal and has constant length

Insertion

$op\ (p \,\hat{}\, q) \simeq (op\ p) \,\hat{}\, q$	
$p \,\hat{}\, (q \,\hat{}\, r) \simeq (p \,\hat{}\, q) \,\hat{}\, r$	$@ \,\hat{}\, p \simeq p$
$p \,\hat{}\, (q \mid r) \simeq (p \,\hat{}\, q) \mid (p \,\hat{}\, r)$	$(p \mid q) \,\hat{}\, r \simeq (p \,\hat{}\, r) \mid (q \,\hat{}\, r)$
$p \,\hat{}\, @ \simeq p$	is generally false
$p \,\hat{}\, (q \,\&\, r) \simeq (p \,\hat{}\, q) \,\&\, (p \,\hat{}\, r)$	is generally false
$(p \,\&\, q) \,\hat{}\, r \simeq (p \,\hat{}\, r) \,\&\, (q \,\hat{}\, r)$	is generally false
$p \,\hat{}\, (op\ q) \simeq (p \,\hat{}\, op)\ q$	the right hand side is nonsense

$@ \,\hat{}\, p$ is equivalent to p since we had an strong equal sign '=' in the corresponding relation of values. $p \,\hat{}\, @$ is not equivalent to p, since the expression $u \,\hat{}\, @$ is less defined than u; here, only '$\acute{=}$' holds.

Iteration

$p^* \simeq [] \mid p+$	
$p+ \simeq p^* \,\&\, !\,[] \simeq [p] \,.\, p^* \simeq p^* \,.\, [p]$	
$(p \,\&\, q)^* \simeq p^* \,\&\, q^*$	
$(p \,\&\, q)+ \simeq p+ \,\&\, q+ \simeq p^* \,\&\, q+ \simeq p+ \,\&\, q^*$	
$p^* \,.\, p^* \simeq p^*$	
$p+ \,.\, p^* \simeq p^* \,.\, p+ \simeq p+$	
$(p^*)^* \simeq p^*$	is generally false

5.9. An alternative semantics of pattern matching

An alternative semantics of pattern matching produces a list of environments instead of a set:

$P'\ (p)\ r\ v = [r_1, \dots, r_n]$ if $P\ (p)\ r\ v = \{r_1, \dots, r_n\}$

There are different approaches to specify the order of the environments in the list:

1) The order is not specified i.e. it is determined by an algorithm whose properties are unknown to the user. A similar approach may be found in some programming languages where the order of the evaluation of arguments of function calls is left unspecified. The user is then restricted to programs where the concrete order is immaterial.

2) The order is specified by the semantics of Trafola and is known to the user, but cannot be changed.
3) The order may be influenced by the user. He/She must essentially define an order for the alternatives produced by the pattern operators ′|′, ′.′, and ′ˆ′ since they are the only sources of non-determinism, and define how two lists are to be combined by an operation corresponding to ′⊕′. We shall now enumerate the several possibilities to do so if no random algorithm is used.

   For the ′|′ operator, there are two possibilities: with ′p | q′, the environments produced by *p* precede those produced by *q* or vice versa. But both ways are essentially equal since the second way corresponds to ′q | p′ handled in the first way.

   For the dot operator, there are also two possibilities: shortest and longest match. Assume ′p . q′ is matched against a sequence; then *p* is matched against subsequences increasing from [] to the whole sequence with shortest match, whereas the order is reversed with longest match.

   The matches of a pattern ′p ˆ q′ may be ordered in four ways aligned along two dimensions: top-down versus bottom-up and left-to-right versus right-to-left.

   Example:

   v = op [1, 2] may be decomposed by four cuts:
   td lr: @ ˆ v, op @ ˆ [1, 2], op [@, 2] ˆ 1, op [1, @] ˆ 2
   td rl: @ ˆ v, op @ ˆ [1, 2], op [1, @] ˆ 2, op [@, 2] ˆ 1
   bu lr: op [@, 2] ˆ 1, op [1, @] ˆ 2, op @ ˆ [1, 2], @ ˆ v
   bu rl: op [1, @] ˆ 2, op [@, 2] ˆ 1, op @ ˆ [1, 2], @ ˆ v

   Thus, we obtain 8 possibilities (2 from ′.′ times 4 from ′ˆ′). The greatest influence is likely to result from the decision top-down or bottom-up.

We shall use the first possibility - at least for the moment - since it is the simplest one for a language designer. In our examples, however, we use a shortest match, top-down, left-to-right strategy.

Once an order is fixed, the ′select′ function may be completely specified by select $[r_1, \dots, r_n] = r_1$.

6. The transformation language

This section contains a revision and great extension of the corresponding section of [1]. It deals with expressions, functions, and declarations.

6.1. Syntax and informal meaning of expressions

Expressions are another important syntactic class of Trafola besides patterns. They denote the two basic constructs in functional languages: function application and functional abstraction, and incorporate special forms for the application of basic operations like concatenation and insertion, and various kinds of syntactic sugar.

Syntax

′e′ and ′f′ stand for expressions, ′p′ for patterns, and ′d′ for definitions, a syntactic class introduced below.

(1)	@	hole
(2)	/c	constant name
(3)	op	operator name
(4)	A	variable name
(5)	$[e_1, \dots, e_n]$	sequence expression
(6)	f e	application of functions or construction of trees
(7)	$e_1 . e_2$	concatenation
(8)	e_1 ˆ e_2	insertion
(9)	$e_1 = e_2$	check on equality
(10)	e_1 != e_2	check on inequality
(11)	$f_1 ; f_2$	functional composition
(12)	$f_1 \mid f_2$	alternative function definitions
(13)	{ p => e }	simple abstraction
(14)	{ p => **all** e }	′all′ abstraction

(15)	**let** d **in** e **end**	local definition
(16)	**letrec** d **in** e **end**	local recursive definition
(17)	**matches** p	matching predicate
(18)	**case** e **of** e' **end**	'case' expression
(19)	**if** e **then** e_1 **else** e_2 **end**	conditional expression

As with patterns, syntactic ambiguities may be solved by parentheses '()' and priorities. In our examples, we shall use some specific order of priorities that is somewhat arbitrary, but globally seems to be useful. In the following list, the operators are ordered by decreasing priority. Under each operator, its associativity is given: 'L' means the operator associates to the left whereas 'A' means that the operator is semantically associative. 'B' means that the operator is only binary i.e. '$e_1 = e_2 = e_3$' is not allowed.

' '	'.'	'^'	'=', '!='	';'	'matches'	'|'	','
L	A	A	B	A	-	A	A

' ' is the operator denoting application. Comma, the operator with least priority, may only occur inside brackets '[' ']'.

The expression in the construct 'p **where** e' and the pattern in '**matches** p' cannot be arbitrary; they must not contain alternatives at their top level. Except this, they extend to the right as much as possible.

Examples:

p | q **where** e | f = p | (q **where** e) | f
{ p **where** F X = G X ^ H . L => e } = {(p **where** ((F X) = (G X) ^ (H . L))) => e}

6.2. Semantics of expressions

The meaning of the constructs (17) through (19) is specified by giving equivalent constructs:

matches p	→ { p => /true } | { _ => /false }
case e **of** e' **end**	→ e' e
if e **then** e_1 **else** e_2 **end**	→ **case** e **of** {/true => e_1} | {/false => e_2} **end**
thus:	→ ({/true => e_1} | {/false => e_2}) e

The meaning of the other constructs is defined by means of a semantic function

E: Expression → Env → Value

using the alternative semantics P' of patterns returning a list of environments, and semantic functions D and R for definitions with

D, R: Definition → Env → Env

where R is used for recursive definitions and D for the other ones.

(1) E (@) r = @
(2) E (/c) r = /c
(3) E (op) r = op

To be precise, we had to distinguish between the expressions '@', '/c', 'op' and their respective values.

(4) E (A) r = if r (A) = unbound then error else r (A)

The value A is bound to, is looked up in the environment.

(5) E ([e_1, ... , e_n]) r = [E (e_1) r, ... , E (e_n) r]
(6) E (f e) r = apply (E (f) r, E (e) r)

using the 'apply' operation of chapter 3.5.

(7) E (e_1 . e_2) r = E (e_1) r . E (e_2) r
(8) E (e_1 ^ e_2) r = E (e_1) r ^ E (e_2) r

Naturally, the operators '.' and '^' in expressions denote the corresponding semantic operation on values.

(9) E ($e_1 = e_2$) r = if Eq (E (e_1) r, E (e_2) r) then /true else /false
(10) E (e_1 != e_2) r = if Eq (E (e_1) r, E (e_2) r) then /false else /true

where Eq (u, v) = if u or v = fail then fail else (u = v)

(11) E (f_1 ; f_2) r = λx. apply (E (f_2) r, apply (E (f_1) r, x))

If f_1 and f_2 evaluate to functions (or operators) F_1 resp. F_2, the expression 'f_1 ; f_2' denotes the function that first applies F_1 and then F_2 to its argument.

(12) E (f_1 | f_2) r = λx. if apply (E (f_1) r, x) = fail

then apply (E (f_2) r, x) else apply (E (f_1) r, x)

If f_1 and f_2 evaluate to functions, ′f_1 | f_2′ denotes the function that first tries to apply the first function, and if it is not defined, applies the second one.

(13) E ({p => e}) r = λx. if s = ∅ then fail else E (e) (r + select s)
where s = P’ (p) r x

The construct ′p => e′ denotes the function matching the pattern *p* against its argument in the environment *r*, and then evaluating the expression *e* in the environment *r* superposed by one of the new environments generated by the pattern match. Trafola is statically scoped since the environment of the abstraction is directly incorporated into the resulting functional value.

(14) E ({p => **all** e}) r = λx. [E (e) (r + r_1), ... , E (e) (r + r_n)]
where [r_1, ... , r_n] = P’ (p) r x

′{p => **all** e}′ denotes the function matching the pattern *p* against its argument in *r*, and then evaluating the expression *e* in *r* superposed by all new environments in turn, thus forming a sequence.

(15) E (**let** d **in** e **end**) r = E (e) (r + D (d) r)

The expression *e* is evaluated in *r*, superposed by a new environment produced by the definition *d*.

(16) E (**letrec** d **in** e **end**) r = E (e) (r + R (d) r)

The definition *d* is here evaluated recursively.

6.3. Definitions

Definitions are used in ′let′ and ′letrec′ expressions, and later in declarations at the top level of a Trafola interpreter. They produce environments and have the general structure:

$p_{11} \cdots p_{1m_1} = e_1$ **and** ... **and** $p_{n1} \cdots p_{nm_n} = e_n$ ($n \geq 1$, $m_i \geq 1$)

where p_{ij} are patterns and e_i expressions.

The meaning is defined by transforming them into a standard form ′p = e′:

$p_1 \cdots p_m = e$ → p_1 = { p_2 => ... { p_m => e }...}
$p_1 = e_1$ **and** ... **and** $p_n = e_n$ → [p_1, ... , p_n] = [e_1, ... , e_n]

Examples:

I X = X → I = {X => X}
First [A, _] = A → First = { [A, _] => A }
Even N = **if** N = 0 **then** /true **else** Odd (Sub N 1) **end** **and**
Odd N = **if** N = 0 **then** /false **else** Even (Sub N 1) **end**
→ [Even, Odd] = [{N => **if** ... **end**}, {N => **if** ... **end**}]

It is easy to specify the semantics of a standardized definition ′p = e′:

D: Definition → Env → Env,
D (p = e) r = if s = ∅ then error else select s
where s = P’ (p) r (E (e) r)

The expression is evaluated in the environment *r*, and the resulting value is matched against the pattern *p*.

Examples:

D (I = {X => X}) r = <I → λx.x>
D (A = B **and** B = A) <A → 1, B → 2> = <A → 2, B → 1>
D (add [A, B] = X) <X → add [1, 2]> = <A → 1, B → 2>

Now we are going to define the environment R (d) r resulting from recursively evaluating *d*. If we knew the result r’, then d should be evaluated in r + r’, i.e. the result is an environment r’ with the property r’ = D (d) (r + r’). Such an environment is a fixed point of the mapping λr’. D (d) (r + r’), and may be obtained by the fixed point combinator ′Y′:

R: Definition → Env → Env, R (d) r = Y (λr’. D (d) (r + r’))

Probably, the generality of a recursive definition will be restricted by some context conditions.

6.4. Top level input

An interactive top level of a Trafola interpreter will accept declarations extending the top level environment and expressions that are evaluated.

Input ::=

dec d	where *d* is a definition evaluated non-recursively
rec d	where *d* is a definition to be evaluated recursively
e	where *e* is an expression

The top level environment is initialized with predefined functions and updated by each input. As a response to the user, each input causes an output that we don't further specify here.

Semantics of input:

I: Input $\rightarrow$ Env $\rightarrow$ (Env X Output)

I (**dec** d) r = (r + r', print r') where r' = D (d) r
I (**rec** d) r = (r + r', print r') where r' = R (d) r
I (e) r = (r + <It $\rightarrow$ v>, print v) where v = E (e) r

The value of the expression is bound to the variable 'It' for further use. This is similar to ML.

The initial top level environment could contain some predefined functions:

init = <
Add $\rightarrow$ λu. λv. if u and v are numbers then u + v else error,
etc. >

The definedness predicate proposed in [4], can be defined in Trafola:

Def F = {X => {_ => /true} (F X)} | {_ => /false}

If 'F X' returns 'fail', this value is propagated such that the function to the left of '|' returns 'fail' and the second function is applied.

7. Conclusion

We gave a definition of a part of Trafola, and left open some points worth to be discussed:

1) Concrete syntax: The concrete syntax described here is only a proposal. Details such as priorities may be discussed further.
2) Matching strategy: Shall it be alterable by the user? If not, which of the 8 possibilities is to be chosen?
3) Handling of failures: This is actually done by the '|' functional and the special value 'fail'. The details of the propagation of 'fail' are not yet quite clear (see 'apply').
4) Handling of non-determinism: A more general mechanism than '=>' and '=> **all**' could be developed. This would influence the handling of failures, and is also likely to interfere with the next topic:
5) Evaluation strategy: Call by value or lazy evaluation? The choice of an evaluation strategy interferes with the handling of non-determinism and failures.
6) Syntactic types: They are not included here, but were proposed in [1] where names for syntactic sorts could be declared by tree grammars and used in patterns. A generalization of this are recursive patterns that may be bound by names, stored somewhere, used as parameters etc.
7) Typing: A polymorphic type discipline should be developed, and the advantages and drawbacks of a type system should be discussed.

Examples of functions written in Trafola will be given in another paper. The topics (6) and (7) will also be discussed further there.

8. Appendix: Summary of semantic clauses

At last, we give a summary of the semantic clauses introduced in this paper.

Standard meaning of letters:

d	definitions

e, f	expressions
p, q	patterns
r	environments
u, v, w	values
/c	constants
op	operators
A	variables

Semantic functions:

At:	Atomic pattern $\rightarrow$ Value $\rightarrow$ Bool
P:	Pattern $\rightarrow$ Env $\rightarrow$ Value $\rightarrow 2^{\mathrm{Env}}$
E:	Expression $\rightarrow$ Env $\rightarrow$ Value
R, D:	Definition $\rightarrow$ Env $\rightarrow$ Env
I:	Input $\rightarrow$ Env $\rightarrow$ (Env X Output)

At (@) v = (v = @)
At (/c) v = (v = /c)
At (op) v = (v = op)
At (_) v = ($v \neq$ fail)

Semantics of patterns:

(0)	P (p) r fail	$= \varnothing$ for all patterns p and environments r
(1)	P (at) r v	= if At (at) v then E else $\varnothing$
(2)	P (% A) r v	= if r(A) = unbound then error else if r(A) = v then E else $\varnothing$
(3)	P (A) r v	= {<A $\rightarrow$ v>}
(4)	P (p $\mid$ q) r v	= P (p) r v $\cup$ P (q) r v
(5)	P (p & q) r v	= P (p) r v $\oplus$ P (q) r v
(6)	P (!p) r v	= if (P (p) r v = $\varnothing$) then E else $\varnothing$
(7)	P ({p}) r v	= if (P (p) r v $\neq \varnothing$) then E else $\varnothing$
(8)	P (p q) r v	= if (v = op w) then P (p) r op $\oplus$ P (q) r w else $\varnothing$
(9)	P ([p_1, ... , p_n]) r v	= if v = [v_1, ... , v_n] then P (p_1) r v_1 $\oplus$ ... $\oplus$ P (p_n) r v_n else $\varnothing$
(10)	P (p*) r v	= if v = [v_1, ... , v_n] then P (p) r v_1 $\oplus$ ... $\oplus$ P (p) r v_n else $\varnothing$
(11)	P (p+) r v	= if v = [v_1, ... , v_n] and v $\neq$ [] then P (p) r v_1 $\oplus$ ... $\oplus$ P (p) r v_n else $\varnothing$
(12)	P (p . q) r v	$= \bigcup_{\text{u, w with u . w = v}}$ P (p) r u $\oplus$ P (q) r w
(13)	P (p ^ q) r v	$= \bigcup_{\text{u, w with u ^ w = v}}$ P (p) r u $\oplus$ P (q) r w
(14)	P (p **where** e) r v	= { r' in P (p) r v $\mid$ E (e) (r + r') = /true }

Semantics of expressions:

matches p	$\rightarrow$ { p => /true } $\mid$ { _ => /false }
case e **of** e' **end**	$\rightarrow$ e' e
if e **then** e_1 **else** e_2 **end**	$\rightarrow$ **case** e **of** {/true => e_1} $\mid$ {/false => e_2} **end**
thus:	$\rightarrow$ ({/true => e_1} $\mid$ {/false => e_2}) e

(1)	E (@) r	= @
(2)	E (/c) r	= /c
(3)	E (op) r	= op
(4)	E (A) r	= if r (A) = unbound then error else r (A)
(5)	E ([e_1, ... , e_n]) r	= [E (e_1) r, ... , E (e_n) r]
(6)	E (f e) r	= apply (E (f) r, E (e) r)
(7)	E (e_1 . e_2) r	= E (e_1) r . E (e_2) r

(8) $E\,(e_1 \,\hat{}\, e_2)\,r \quad = \quad E\,(e_1)\,r \,\hat{}\, E\,(e_2)\,r$

(9) $E\,(e_1 = e_2)\,r \quad = \quad \text{if } Eq\,(E\,(e_1)\,r, E\,(e_2)\,r) \text{ then } /\text{true} \text{ else } /\text{false}$

(10) $E\,(e_1 \mathrel{!=} e_2)\,r \quad = \quad \text{if } Eq\,(E\,(e_1)\,r, E\,(e_2)\,r) \text{ then } /\text{false} \text{ else } /\text{true}$

where $Eq\,(u, v) = \text{if } u \text{ or } v = \text{fail then fail else } (u = v)$

(11) $E\,(f_1 \,;\, f_2)\,r \quad = \quad \lambda x.\ \text{apply}\,(E\,(f_2)\,r, \text{apply}\,(E\,(f_1)\,r, x))$

(12) $E\,(f_1 \mid f_2)\,r \quad = \quad \lambda x.\ \text{if apply}\,(E\,(f_1)\,r, x) = \text{fail}$
$\qquad\qquad \text{then apply}\,(E\,(f_2)\,r, x) \text{ else apply}\,(E\,(f_1)\,r, x)$

(13) $E\,(\{p \Rightarrow e\})\,r \quad = \quad \lambda x.\ \text{if } s = \varnothing \text{ then fail else } E\,(e)\,(r + \text{select } s)$
$\qquad\qquad \text{where } s = P'\,(p)\,r\,x$

(14) $E\,(\{p \Rightarrow \textbf{all } e\})\,r \quad = \quad \lambda x.\ [E\,(e)\,(r + r_1), \ldots, E\,(e)\,(r + r_n)]$
$\qquad\qquad \text{where } [r_1, \ldots, r_n] = P'\,(p)\,r\,x$

(15) $E\,(\textbf{let } d \textbf{ in } e \textbf{ end})\,r \quad = \quad E\,(e)\,(r + D\,(d)\,r)$

(16) $E\,(\textbf{letrec } d \textbf{ in } e \textbf{ end})\,r \quad = \quad E\,(e)\,(r + R\,(d)\,r)$

Semantics of definitions:

$p_1 \ldots p_m = e \quad \to \quad p_1 = \{\, p_2 \Rightarrow \ldots \{\, p_m \Rightarrow e \,\} \ldots \}$

$p_1 = e_1 \textbf{ and } \ldots \textbf{ and } p_n = e_n \quad \to \quad [p_1, \ldots, p_n] = [e_1, \ldots, e_n]$

$D\,(p = e)\,r = \text{if } s = \varnothing \text{ then error else select } s$
$\qquad \text{where } s = P'\,(p)\,r\,(E\,(e)\,r)$

$R\,(d)\,r = Y\,(\lambda r'.\ D\,(d)\,(r + r'))$

Semantics of input:

$I\,(\textbf{dec } d)\,r \quad = \quad (r + r', \text{print } r') \text{ where } r' = D\,(d)\,r$

$I\,(\textbf{rec } d)\,r \quad = \quad (r + r', \text{print } r') \text{ where } r' = R\,(d)\,r$

$I\,(e)\,r \quad = \quad (r + \langle It \to v \rangle, \text{print } v) \text{ where } v = E\,(e)\,r$

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]

[4] Heckmann, R.: Revision of the Previous Work, [S.1.6-SN-9.0]