PROSPECTRA
an ESPRIT Project

Notes on TRAFOLA, III
Semantics of patterns

Reinhold Heckmann

Universität des Saarlandes
6600 Saarbrücken
Bundesrepublik Deutschland

PROSPECTRA Study Note S.1.6-SN-8.0

1987-06-03

Distribution: public

ABSTRACT

In this note, we shall make a proposal for the syntax and semantics of patterns in the transformation language TRAFOLA. First, we shall define and investigate semantic domains for patterns, and consider semantic equivalence of patterns. Finally, some subclasses of patterns e.g. linear patterns, will be introduced.

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. Semantic domains and their properties

1.1. Introduction

In this chapter, we want to introduce the basic semantic domains that allow for the definition of the semantics of patterns. Patterns are considered to be non-deterministic. When applied to a value, they yield a set of environments binding names to values.

Although patterns are not yet formally introduced, we give an example for non-deterministic matching. Let *c, d, e* be different constants and let *p* be the pattern

(A: (c | d)*, B: (d | e)*)

It matches all concatenations of a sequence of *c* or *d* constants and a sequence of *d* and *e* constants, and binds the first sequence to the name *A* and the second sequence to the name *B*. The pattern is not deterministic since an occurrence of *d* could belong to the first sequence as well as to the second one.

Value	Resulting set of environments
()	$\{[A \to (), B \to ()]\}$
(e, c)	$\varnothing$ (the pattern does not match)
(c, d, e)	$\{[A \to (c, d), B \to e], [A \to c, B \to (d, e)]\}$

The semantic function *P* for patterns will therefore have the type

$P: Pat \to Val \to 2^{Name \to Val}$

It will be defined by operations on the set $2^{Name \to Val}$ that we shall define by stepwise extending operations in *Val* and *Name − > Val*.

1.2. Superposition of values and environments

Definition (1.1)

Let *Val* be a set of values, and *Name* a set of names. Then we define

Val’	$= Val \cup \{\perp\}$	Val"	$= Val' \cup \{T\}$
Env’	$= Name \to Val'$	Env"	$= Name \to Val"$
Sem’	$= 2^{Env'}$	Sem"	$= 2^{Env"}$

Here, *Val* denotes the set of well formed values as introduced in [2], but *Val’* is a new set different from the set of general values *Val’* of [2].

Meaning of $\perp$ and T:

Let *a* be an environment and *n* a name.

$a(n) = \perp$	means: n is unbound
$a(n) = v \in Val$	means: n is bound to the value v
$a(n) = T$	means: n was bound several times to inconsistent values

Val" is considered a lattice where $\perp$ is the least element, all ordinary values (members of *Val*) are incomparable, and T is the greatest element. We denote the union in this lattice by $'+'$. Proposition (1.2) below follows immediately from lattice theory:

Proposition (1.2)

For all u, v, w in Val" holds:

$u + v = v + u \qquad u + (v + w) = (u + v) + w$

$u + \perp = u \qquad u + T = T \qquad u + u = u$

$u + v = \perp$ iff $u = \perp$ and $v = \perp$

The operation $'+'$ may be lifted to environments:

Definition (1.3)

For environments a, b in Env" = Name $\rightarrow$ Val"
let a + b = λn. (a (n) + b (n))
Empty environment e := λn. $\perp$
Domain of an environment: dom (a) := { n $\in$ Name | a(n) $\neq \perp$ }

Example:

Let A, B, C be names and v and w different values.
a = [A $\rightarrow \perp$, B $\rightarrow$ v, C $\rightarrow$ v] b = [A $\rightarrow$ v, B $\rightarrow$ v, C $\rightarrow$ w].
Then a + b = [A $\rightarrow$ v, B $\rightarrow$ v, C $\rightarrow$ T]

A is unbound in *a* and bound to *v* in *b*, thus *A* is bound to *v* in *a + b*. Since *B* is bound to *v* in both *a* and *b*, it is also bound to *v* in *a + b*. *C* is bound to different values by *a* and *b*, thus *C* is bound to T in *a + b*.

The properties of '+' for values easily translate to those for environments:

Proposition (1.4)

For all a, b, c in Env" holds:

a + b = b + a	a + (b + c) = (a + b) + c
a + e = a	a + a = a
a + b = e iff a = e and b = e	
dom (e) = $\emptyset$	dom (a + b) = dom (a) $\cup$ dom (b)

The mapping *dom* is therefore a homomorphism from (Env", +, e) to (2^{Name}, $\cup$, $\emptyset$).

Later, we shall need the following lemma:

Lemma (1.5)

For all a, b in Env' holds:
if dom (a) $\subseteq$ dom (b) and a + b in Env', too, then a + b = b.

Proof:
We shall show a(n) + b(n) = b(n) for all names n.
Case n not in dom (a): a(n) + b(n) = $\perp$ + b(n) = b(n)
Case n in dom (a):
Then n is in dom (b), too, and thus a(n) and b(n) are in Val, i.e. ordinary values. If a(n) and b(n) were different, their sum (a+b)(n) would be T, and a + b would not be in Env'. Thus, they are equal, and
a(n) + b(n) = b(n) + b(n) = b(n).

1.3. Operations on sets of environments

Definition (1.6)

Special elements of Sem":	$\emptyset$ and E := {e}	
Operations on Sem":	Union $\cup$	
Superposition:	s + t	:= { a + b | a $\in$ s, b $\in$ t }
Reduction:	R (s)	:= s $\cap$ Env'
Unification:	s $\oplus$ t	:= R (s + t)

If two sets of environments are unified, they are first superposed by pairwise merging the contained environments. Then all environments that result from merging inconsistent environments and thus map some name to T, are eliminated by the reduction *R*.

Example:

$\{[A \rightarrow \perp, B \rightarrow v]\ (1),\ [A \rightarrow v, B \rightarrow w]\ (2)\} \ \oplus$
$\{[A \rightarrow v, B \rightarrow v]\ (3),\ [A \rightarrow w, B \rightarrow \perp]\ (4)\} =$
$R\ (\{[A \rightarrow v, B \rightarrow v]\ (1{+}3),\ [A \rightarrow w, B \rightarrow v]\ (1{+}4),$
$[A \rightarrow v, B \rightarrow T]\ (2{+}3),\ [A \rightarrow T, B \rightarrow w]\ (2{+}4)\}) =$
$\{[A \rightarrow v, B \rightarrow v],\ [A \rightarrow w, B \rightarrow v]\}$

The properties of union are well known:

Proposition (1.7)

For all s, t, u in Sem" holds:
$s \cup t = t \cup s$ $\quad$ $s \cup (t \cup u) = (s \cup t) \cup u$
$s \cup \varnothing = s$ $\quad$ $s \cup s = s$
$s \cup t = \varnothing$ iff $s = \varnothing$ and $t = \varnothing$
if s and t are in Sem', then $s \cup t$ is also in Sem'
if s and t are finite, then $s \cup t$ is finite, too, and $|s \cup t| \le |s| + |t|$

The properties of '+' are similar:

Proposition (1.8)

For all s, t, u in Sem" holds:
$s + t = t + s$ $\quad$ $s + (t + u) = (s + t) + u$
$s + \varnothing = \varnothing$ $\quad$ $s + E = s$
$s + t = \varnothing$ iff $s = \varnothing$ or $t = \varnothing$
$s + t = E$ iff $s = E$ and $t = E$
if s and t are finite, then $s + t$ is finite, too, and $|s + t| \le |s| \cdot |t|$

Proof:

All properties except '*s + t = E iff s = E and t = E*' are obvious, they follow from set theory and corresponding properties of Prop. (1.4).
If s and t equal E, $s + t = E + E = E$ holds.
For the opposite direction, assume $s + t = E = \{e\}$. Thus $e \in s + t$, hence there exist a in s and b in t such that $a + b = e$. By Prop. (1.4), $a = b = e$ follows and thus e in s and e in t. If s contained an element c different from e, then s + t would contain $c + e = c \ne e$. Therefore $s = \{e\}$ and analogously $t = \{e\}$ hold.

The operation '+' on sets of environments does not share all properties of the corresponding operation on environments. Idempotence is missing, a fact that will pose some problems in the next chapter.

Proposition (1.9)

For all s in Sem", $s + s \supseteq s$ holds,
but $s + s = s$ is not generally true.

Proof:

The first statement is fairly obvious: if $a \in s$, then $a = a + a \in s + s$.
Example for $s + s \ne s$:
Let $a = [A \rightarrow v, B \rightarrow \perp]$ and $b = [A \rightarrow \perp, B \rightarrow w]$.
Then $c = a + b = [A \rightarrow v, B \rightarrow w]$,
and $\{a, b\} + \{a, b\} = \{a + a, a + b, b + a, b + b\} = \{a, b, c\} \ne \{a, b\}$.

Relationships between '+' and '$\cup$' resp. '$\subseteq$':

Proposition (1.10)

For all s, t, t', t_i in Sem" holds:
if $t \subseteq t'$ then $s + t \subseteq s + t'$ $\qquad s + \bigcup_i t_i = \bigcup_i (s + t_i)$

The proof is not hard and left to the reader.

The properties of the reduction *R* follow directly from set theory:

Proposition (1.11)

For all s, t, s_i in Sem" holds:

$R(s) \subseteq s \qquad R(R(s)) = R(s) \qquad R(\varnothing) = \varnothing \qquad R(E) = E$

$s \subseteq t$ implies $R(s) \subseteq R(t) \qquad R(\bigcup_i s_i) = \bigcup_i R(s_i)$

There is a relationship between *R* and '+':

Lemma (1.12): For all s, t in Sem": $R(s + R(t)) = R(s + t)$

Proof:

$\subseteq$: $R(t) \subseteq t$ implies $s + R(t) \subseteq s + t$ and thus
$R(s + R(t)) \subseteq R(s + t)$ by Prop. (1.10) and (1.11)

$\supseteq$: Let a be in $R(s + t) = (s + t) \cap Env'$.
Thus $a(n) \neq T$ for all names n, and $a = b + c$ where b in s and c in t.
If there were a name n such that $c(n) = T$
then $a(n) = b(n) + c(n) = T$ would result in a contradiction.
Thus c is in Env', hence c is in R(t) and $a = b + c$ in $s + R(t)$.
Since a is in Env', a is in $R(s + R(t))$.

1.4. Unification of sets of environments

Superposition '+' and reduction *R* are only auxiliary operations, the really interesting operation is unification '⊕'. Its properties are similar to those of '+':

Proposition (1.13)

For all s, t, t', t_i, u in Sem" holds:
$s \oplus t$ is always in Sem'
$s \oplus t = t \oplus s \qquad s \oplus (t \oplus u) = (s \oplus t) \oplus u \qquad s \oplus \varnothing = \varnothing$
For all s, t in Sem': $\quad s \oplus E = s \qquad s \oplus t = E$ iff $s = E$ and $t = E$

if s and t are finite, then $s \oplus t$ is finite, too, and |s ⊕ t| ≤ |seally interesting operation is unification '⊕'. Its properties are similar to those of '+':

Proposition (1.13)

For all s, t, t', t_i, u in Sem" holds:
$s \oplus t$ is always in Sem'
$s \oplus t = t \oplus s \qquad s \oplus (t \oplus u) = (s \oplus t) \oplus u \qquad s \oplus \varnothing = \varnothing$
For all s, t in Sem': $\quad s \oplus E = s \qquad s \oplus t = E$ iff $s = E$ and $t = E$

if s and t are finite, then $s \oplus t$ is finite, too, and $|s \oplus t| \leq |s| \cdot |t|$

if $t \subseteq t'$ then $s \oplus t \subseteq s \oplus t' \qquad s \oplus \bigcup_i t_i = \bigcup_i (s \oplus t_i)$

For all s in Sem', $s \oplus s \supseteq s$ holds, but $s \oplus s = s$ is not generally true.
The following statement is also not true: $s \oplus t = \varnothing$ iff $s = \varnothing$ or $t = \varnothing$

Proof:

$s \oplus t = R(s + t)$ is obviously in Sem'
Commutativity is trivial.
Associativity: $s \oplus (t \oplus u) = R(s + R(t + u))$
$= R(s + (t + u))$ by Lemma (1.12)

$= R ((s + t) + u)$ by associativity of $'+'$
$= R (R (s + t) + u)$ by Lemma (1.12) again and commutativity of $'+'$
$= (s \oplus t) \oplus u.$

$s \oplus \emptyset = R (s + \emptyset) = R(\emptyset) = \emptyset$

$s \oplus E = R (s + E) = R (s) = s$ since s in Sem'

The proof of the property concerning equality with *E* is analogous to the proof of the corresponding property of $'+'$. For both properties, the restriction to *Sem'* is essential:

$\{[A \to T]\} \oplus E = \emptyset$ $\{e, [A \to T]\} \oplus \{e, [A \to T]\} = E$

Finiteness, monotony, and distributivity are direct consequences of the properties of $'+'$ and *R*.

$s \oplus s \supseteq s$: a in s implies $a = a + a$ in $s + s$.
Since s in Sem', a is in Env' and thus in $s \oplus s$.

The example for $s \oplus s \neq s$ is the same as for $s + s \neq s$.

Example for the last claim: let v and w be different values.
$\{[A \to v]\} \oplus \{[A \to w]\} = R (\{[A \to T]\}) = \emptyset.$

Now we shall state and prove a lemma that looks rather technically, but will be needed later.

Lemma (1.14)

Let s and t be elements of Sem' such that for all a in s and b in t
$dom (a) \subseteq dom (b)$ holds. Then $s \oplus t \subseteq t$.

Proof:

Let c be a member of $s \oplus t$. Thus c is in Env', and $c = a + b$ where a in s and b in t. Then $dom (a) \subseteq dom (b)$ holds due to the precondition, and a and b are members of Env'. By Lemma (1.5), we obtain that $c = a + b$ equals b, and is thus in t.

To prove $s \oplus s = s$ for a set *s*, we can use the lemma if we have $dom (a) \subseteq dom (b)$ for all environments *a* and *b* in *s*. Since *a* and *b* may be permuted here, this precondition means that the domains of all environments contained in *s* are equal.

Definition (1.15)

A set s in Sem" is called uniform iff all environments in it have the same domain: for all a, b in s: $dom (a) = dom (b)$.

Proposition (1.16)

If s and t are uniform, then $R(s)$, $s + t$, and $s \oplus t$ are also uniform.
If s is uniform and in Sem', then $s \oplus s = s$ holds.

Proof:

R(s): obvious.

$s + t$: let c, c' be in $s + t$.
Then there are a, a' in s and b, b' in t such that $a + b = c$ and $a' + b' = c'$.
Hence $dom (c) = dom (a + b) = dom (a) \cup dom (b) =$
$dom (a') \cup dom (b') = dom (c')$.

$s \oplus t$: directly from $'+'$ and *R*.

$s \oplus s \supseteq s$ is already known.

$s \oplus s \subseteq s$ is a direct consequence of Lemma (1.14).

1.5. Iterated unification

The results of this section will be important to investigate the semantics of iterated patterns, e.g. p*.
At first, we introduce a notion of iterated unification:

Definition (1.17): For all s in Sem': $s^1 = s$ $s^{n+1} = s \oplus s^n$

Let *s* be a fixed member of *Sem'* in the sequel. From Prop. (1.13), we know that $s^1 = s$ is a subset of s^2

$= s \oplus s$, and that the operation $\oplus$ is monotone such that $s^n \subseteq s^{n+1}$ follows by induction. Consider the chain

$$s^1 \subseteq s^2 \subseteq s^3 \cdots$$

Either all inclusions are proper, then the chain infinitely ascends, or there is an exponent *i* such that $s^i = s^{i+1}$, then $s^j = s^i$ for all $j \geq i$ holds by induction. We call the chain finite in the second case, and we are interested in a criterion for finite chains and in an estimation for the size of the exponent *i* where the chain ends.

All sets in the chain are obviously subsets of *Sem'*, thus the chain could be proved to be finite if both *Name* and *Val* were finite. But *Val* is infinite since it contains arbitrarily long sequences of atomic values. Fortunately, there is a better criterion not depending on *Val*.

Let *M* and *N* be two sets of names with $M \subseteq N$ such that for all environments *a* in *s* holds:

$M \subseteq \text{dom}(a) \subseteq N$

Such sets always exist e.g. $M = \emptyset$ and $N = Name$. Let *m* and *n* be the cardinalities of *M* and *N* respectively. We shall prove in the sequel that the chain is finite if the set *N* is finite, and the difference *n - m* giveti -0.6c
$\subseteq$: given a in s, construct b := λn. if n in M then a(n) else $\perp$
Then dom (b) is M, thus b is in s^0, and dom (b) $\subseteq$ dom (a), hence a = a + b due to Lemma (1.5).

Now let

$$t_k := s^{k+1} - s^k \quad \text{for all } k \geq 0$$

be the set of environments occurring in s^{k+1}, but not in s^k. With these notions, we prove the following lemma:

for all $k \geq 0$: for all a in t_k is $|\text{dom}(a)| > m + k$

Proof by induction on k:

Case k = 0: $t_0 = s^1 - s^0 = s - s^0$.

Let a be in t_0. Then a is in s whence $M \subseteq \text{dom}(a)$, but a is not in s^0 whence $M \neq \text{dom}(a)$. Thus, $|\text{dom}(a)| > m = m + 0$.

Step from k to k+1: Let a be in $t_{k+1} = s^{k+2} - s^{k+1} = s \circ s^{k+1} - s \circ s^k$.

Hence a = b + c, where b in s and c in s^{k+1}. If c were in s^k, then a = b + c would be in $s \oplus s^k$. Thus c is in s^{k+1}, but not in s^k, therefore it is in t_k, and by induction hypothesis, $|\text{dom}(c)| > m + k$ holds.

If dom (b) were a subset of dom (c), then a = b + c would equal c by Lemma (1.5) since a, b, and c are all in Env'. But this equality is impossible since c is in s^{k+1}, but a is not.

Therefore dom (b) $\subseteq$ dom (c) is false, and there is a name n in dom (b) that is not a member of dom (c). By Prop. (1.4), dom (a) = dom (b) $\cup$ dom (c) holds, thus

$\text{dom}(a) \supseteq \{n\} \cup \text{dom}(c)$ and $|\text{dom}(a)| \geq 1 + |\text{dom}(c)| > m + k + 1$.

From this auxiliary lemma, we are able to conclude our statement about chains. Let the set *N* be finite, and *n* its cardinality. Consider $t_{n-m} = s^{n-m+1} - s^{n-m}$. For each environment a in this set, $|\text{dom}(a)| > m + n - m = n$ holds, but $\text{dom}(a) \subseteq N$ implies $|\text{dom}(a)| \leq n$. This contradiction proves that t_{n-m} is empty, and thus, $s^{n-m} \supseteq s^{n-m+1}$.

If n-m > 0, we may hence conclude $s^{n-m} = s^{n-m+1}$ since '$\subseteq$' is already known. If n-m = 0, i.e. M equals N, then s is uniform and $s^1 = s^2$ holds by Prop. 2.16.

The results of this paragraph may be summarized to the following lemma:

Lemma (1.18)

For all sets of environments s in Sem' holds:
$s^1 \subseteq s^2 \subseteq s^3 \subseteq \cdots$
If there are two finite sets of names M and N such that
for all environments a in s $M \subseteq dom(a) \subseteq N$ holds,
then this chain is finite and
for all $j \geq i := \max(1, |M - N|)$, $s^j = s^i$ holds.

2. Patterns and their semantics

2.1. Syntax of patterns

Let *Atomic* be a set of atomic patterns containing syntactic sorts such as *Stm* in [1], nullary operators, *()*, *[]*, and wild cards like *'_'*, *' ^ '*, etc. *Name* is a denumerable set of variable names disjoint from the set *Atomic*. *Op* is a set of operators.

(2.1) Syntax of patterns (each line is an alternative)

Pat ::=
  Atomic
  Name
  op Pat
  Pat | Pat
  Pat & Pat
  ! Pat
  Pat , Pat
  Pat □ Pat — where □ is standing for the insertion operators Δ, A, Λ
  Pat Iterator
  { Pat }
Iterator ::= {i .. j} — where i, j are integers with $0 \leq i \leq j \leq \infty$ and $i < \infty$

Alternative paraphrasing of iterators:

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

To resolve ambiguities, patterns may be grouped by round parentheses *( )*. In practice, priorities should be introduced.

2.2. Informal semantics of patterns

Intuitively, a pattern denotes the set of values it matches, and binds some variables to some values, it thus produces environments. Patterns serve to analyze values and to produce environments, whereas expressions serve to synthesize values from subvalues stored in environments.

The basic construct of the future transformation language will be *'pattern => expression'* denoting a mapping that first decomposes its input value by the pattern into subvalues bound to names, and then constructs a new value from these subvalues.

If a name occurs more than once in a pattern such that it matches several subvalues, the whole pattern matches only if these subvalues are equal (unification semantics, see discussion in [1]). Thus there are two reasons to use names: to bind values for later use in an expression, and to restrict the variety of values matched by a pattern by means of unification.

Remember the informal meaning of the several constructs:

Atomic serves to denote certain subsets of values to be matched. No names are bound. Examples are nullary operators, syntactic sorts, or wild cards.

Name A name *A* matches any value and binds *A* to it. Names should never occur unrestricted, but always in connection with another pattern *p* by *A & p*. For this construct *A & p*, we propose an alternative paraphrasing by *A: p*. The construct *A: p* matches as *p* does and binds *A* to the matched value.

op p matches all values *op w* where *w* is matched by *p*

p | q *p or q*, matches any value matched by *p* or *q*

p & q *p and q*, matches any value matched by *p* and *q*

!p *not p*, matches any value that *p* does not match

p, q matches any concatenation of values matched by *p* and *q*

p □ q matches any result of syntactic insertion □ = Δ, A, or Λ of values matched by *p* and *q*. In [1], we had written *p q* instead of *p* Λ *q*, but for the theory, it is better to have a visible operator.

p {i..j} matches any concatenation of *i* through *j* values matched by *p*

{p} unbinding operator, *{p}* matches the same values as *p* does, but does not bind variables, i.e. all bindings implied by *p* are canceled. It is useful in theory, but practically only needed in rare occasions.

The syntactic operators comma, Δ, A, and Λ of patterns are the inverse operations of concatenation ·, Δ, A, and Λ of values.

Examples:

In the examples, we abbreviate the environment [A → v, B → w] by [v | w].
The pattern *(p, q)* applied to value *v* is analogous to the Prolog instruction *append (p, q, v)*.

Pattern	Value	Result
(A, B)	()	{[() | ()]}
(A, B)	(a, b, c)	{[() | (a,b,c)], [a | (b,c)], [(a,b) | c], [(a,b,c) | ()]}
(A, u, B)	(u, v, w)	{[() | (v,w)]}
(A, A, B)	(a,b,a,b,a)	{[() | (a,b,a,b,a)], [(a,b) | a]}
A Λ B	add (v, w)	{[[] | add (v, w)], [add [] | (v, w)], [add ([], w) | v], [add (v, []) | w], [([], add (v, w)) | ()], [(add (v, w), []) | ()], [add ([], v, w) | ()], [add (v, [], w) | ()], [add (v, w, []) | ()]}
A Δ B	add (v, w)	{[add (v, w) | ()], [[] | <add (v, w)>], [add [] | <v, w>], [add ([], []) | <v; w>], [add ([], w) | <v>], [add (v, []) | <w>]}
A Λ B	add (v, w)	{[add (v, w) | ()], [[] | add (v, w)], [add ([], w) | v], [add (v, []) | w], [add ([], []) | (v, w)]}

Differences among *A* Λ *B*, *A* A *B*, and *A* Δ *B*:

- By *A* Λ *B*, the value bound to *A* contains exactly one hole, whereas by *A* Δ *B* and *A* A *B*, it may contain an arbitrary number of holes.
- The value bound to B is grouped with Δ, whereas it is not grouped with Λ or A.

For patterns *p* and *q*, *p* Λ *q* and *p* Δ *<q>* seem to be equivalent; we shall investigate these topics later more closely.

In [1], we have claimed that a pattern preceded by a not operator must not contain names since these names could logically not be bound. But now, we think it is useful to allow names after ′!′ for unification whose binding is canceled by the not operator.

Example: add (A: Exp, B: Exp) & ! add (E: Exp, E: Exp)
applied to add (1, 2) yields {[A → 1, B → 2]}

applied to add (1, 1) yields ∅.

This pattern matches sums with different summands and binds *A* and *B* to these summands. The name *E* is needed to ensure that *A* and *B* are not equal, but *E* is not bound by the whole pattern since the not operator cancels the binding of *E*.

Names under an iterator are another point of discussion. An iteration is merely an abbreviation for a (possibly infinite) alternation of concatenations. Thus names under an iterator should participate in unification.

Examples:

N: (p+)	matches lists of values having shape *p* and binds *N* to the whole list
(N: p)+	matches lists of values having shape *p* where the list is made up from completely identical items; *N* is bound to this item

Pattern	Value	Result
A: (Exp+)	(1, 2)	$\{[A \to (1, 2)]\}$
(A: Exp)+	(1, 2)	∅
(A: Exp)+	(1, 1)	$\{[A \to 1]\}$
(add (A: Exp, A: Exp))+	add (1, 2)	∅
(add (A: Exp, A: Exp))+	(add (1, 1), add (2, 2))	∅
(add (A: Exp, A: Exp))+	(add (1, 1), add (1, 1))	$\{[A \to 1]\}$

If we want to match lists of sums with equal summands where the list items need not be equal, we must use the unbind operator:

{ add (A: Exp, A: Exp) }+	(add (1, 1), add (2, 3))	∅
{ add (A: Exp, A: Exp) }+	(add (1, 1), add (2, 2))	$E = \{\lambda n. \perp\}$

This is the only useful application of the unbind operator in practice.

Four statements about names under iterators:

1) Names under iterators are not very useful.
2) As we shall see later, actual bound names are even dangerous and should be forbidden if the iteration begins with 0.
3) One reason to use names under iterators is to specify lists of items having sub-items that are identical throughout the list, e.g. (N: p)+.
4) The other reason is to restrict the possible items by unification, e.g.
   { add (E: Exp, E: Exp) } +
   This is also the only useful application of the unbind operator.

2.3. 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.

With each atomic pattern, a set of values is associated, called its language *L*. We assume the set *Atomic* and the mapping *L: Atomic* $- > 2^{Val}$ to be given. *Atomic* should contain an element () with L (()) = {()}, an element [] with L ([]) = {[]}, and for theoretic issues, an element *all* is needed with *L (all)* = *Val*. Later, we shall present a proposal for useful atomic patterns.

(2.2) Semantics of Patterns P: Pat → Val → Sem'

(1)	P (at) v	= if v in L (at) then E else ∅	
(2)	P (op p) v	= if (v = op w) then P (p) w else ∅	
(3)	P (A) v	= $\{[A \to v]\}$	for names A
(4)	P (p | q) v	= $P(p)\, v \cup P(q)\, v$	
(5)	P (p & q) v	= $P(p)\, v \oplus P(q)\, v$	
(6)	P (!p) v	= if (P (p) v = ∅) then E else ∅	
(7)	P ({p}) v	= if (P (p) v ≠ ∅) then E else ∅	
(8)	P (p, q) v	= $\bigcup_{u, w \text{ with } u \cdot w = v} P(p)\, u \oplus P(q)\, w$	

(9)	P (p □ q) v	$= \bigcup_{u,\, w \text{ with } u \,\square\, w \,=\, v}$ P (p) u ⊕ P (q) w	□ = Δ, A, Λ
(10)	P (p {i..j}) v	$= \bigcup_{k=i}^{j}$ P (p {k}) v	
(11)	P (p {0}) v	= P (()) v	
(12)	P (p {k}) v	= P (p, p {k-1}) v	if k > 0

The first question is: is *P* computable, i.e. are all operations in the definition above effective? The most dangerous operation is the union in (10) if *j* is ∞. Later, we shall prove that *P* is in fact computable, but now we state only

Lemma (2.3)

If the union in (10) can be reduced to a finite union for any given pattern and value, then the set of environments *P (p) v* is finite for every pattern *p* and value *v*, and if, in addition, the test in (1) is computable, the semantic mapping *P* is also computable.

Proof: (by induction on *p*)

Since only finite values are matched, the unions of (8) and (9) are finite (see [2], chapter 9). *E* and ∅ are finite, and union and unification ⊕ combine finite sets of environments to finite sets again. Thus all resulting sets are finite. The operations ∪ and ⊕ are effective on finite sets, algorithms could be written for them in a functional style if sets are represented as lists.

We shall consider the critical union in (10) later. First, we shall investigate the semantics of pattern more closely.

2.4. Equivalence of patterns

Definition (2.4) Equivalence of patterns:

p ≃ q iff P (p) v = P (q) v for all values v

Equivalence is a semantic property, two equivalent patterns cannot be distinguished by their semantics. Equivalence is a congruence relation since the semantic function *P* for a pattern only depends on the semantics of subpatterns.

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

In the sequel, we shall state and prove equivalences, and disprove conjectures by counterexamples. Most of these examples contain pathologic occurrences of names to be bound. They can be excluded by context conditions for names that will be presented later.

Properties of | and &:

The properties of | and & are a direct consequence of those of ∪ and ⊕ (Prop. 1.7, 1.13). Note that P (all) v = E for all values v.

p | q ≃ q | p	p | (q | r) ≃ (p | q) | r	p | p ≃ p
p & q ≃ q & p	p & (q & r) ≃ (p & q) & r	p & all ≃ p
p & (q | r) ≃ (p & q) | (p & r)		

′p | all ≃ all′ is obviously false if names are bound by *p*.

The property ′p & p ≃ p′ does not hold in general. Consider the following two examples:

1) p = (A: ()) | (B: ())

2) p = (A: ())?, (B: ())?

both match the value () and yield the set {[A → ⊥, B → ()], [A → (), B → ⊥]}. This is the set of example (1.9).

Note that both example patterns suffer from strange usage of names: they produce environments where

A is bound as well as environments where it is not bound.

Let in the following examples *c* and *d* always be different nullary operators.

The opposite distributivity does not hold:
$p \mid (q \,\&\, r) \simeq (p \mid q) \,\&\, (p \mid r)$ is not true.

Example: p = c q = A: c r = d application to value c.

$$\begin{aligned}
&P\,(c \mid ((A{:}\ c) \,\&\, d))\ c \;=\; P\,(c)\ c \,\cup\, (P\,(A{:}\ c)\ c \,\oplus\, P\,(d)\ c)\\
&\;=\; E \,\cup\, (\{[A \rightarrow c]\} \,\oplus\, \emptyset) \;=\; E \cup \emptyset \;=\; E\\
&\text{but}\;\; P\,(\,(c \mid A{:}\ c) \,\&\, (c \mid d)\,)\ c \;=\; (E \,\cup\, \{[A \rightarrow c]\}) \,\oplus\, (E \,\cup\, \emptyset)\\
&\;=\; \{e, [A \rightarrow c]\} \,\oplus\, E \;=\; \{e, [A \rightarrow c]\}.
\end{aligned}$$

The laws of absorption of lattice theory don't hold:
$p \,\&\, (p \mid q) \simeq p$ is not true, $p \mid (p \,\&\, q) \simeq p$ is also not true.

Example: p = c q = A: c application to value c.

$$\begin{aligned}
&P\,(c)\ c = E\\
&P\,(c \,\&\, (c \mid A{:}\ c))\ c \;=\; E \oplus (E \cup \{[A \rightarrow c]\}) \;=\; \{e, [A \rightarrow c]\}\\
&P\,(c \mid (c \,\&\, A{:}\ c))\ c \;=\; E \cup (E \oplus \{[A \rightarrow c]\}) \;=\; \{e, [A \rightarrow c]\}
\end{aligned}$$

Later, we shall be able to prove these laws under certain conditions for the usage of names.

Properties of name binding:

$$\begin{aligned}
&A{:}\ (B{:}\ p) \simeq B{:}\ (A{:}\ p) \qquad\qquad A{:}\ (A{:}\ p) \simeq A{:}\ p\\
&A{:}\ (p \,\&\, q) \simeq (A{:}\ p) \,\&\, q \simeq p \,\&\, (A{:}\ q)\\
&A{:}\ (p \mid q) \simeq (A{:}\ p) \mid (A{:}\ q)
\end{aligned}$$

Proof:

Remember that ′A: p′ is merely an alternative paraphrasing of ′A & p′. Thus, all properties directly follow from those of ′&′ except the second one. It is a consequence of
$\{[A \rightarrow v]\} \oplus \{[A \rightarrow v]\} = \{[A \rightarrow v]\}$.

Properties of operators:

$$op\ (p \mid q) \simeq (op\ p) \mid (op\ q) \qquad\qquad op\ (p \,\&\, q) \simeq (op\ p) \,\&\, (op\ q)$$
$$op\ \{p\} \simeq \{\ op\ p\ \}$$

The proofs are rather straightforward.

Properties of ! and {}:

$$\{\{p\}\} \simeq \{p\} \qquad \{!p\} \simeq\ !p \qquad !\{p\} \simeq\ !p \qquad !!p \simeq \{p\}$$

These four statements are quite obvious, remember that the not operator contains an implicit unbinding by the definition of its semantics. Thus ′$!!p \simeq \{p\}$′ holds, but ′$!!p \simeq p$′ is not generally true.

Both operators are compatible with ′|′:

$$\{p \mid q\} \simeq \{p\} \mid \{q\} \qquad\qquad !\,(p \mid q) \simeq\ !p \,\&\, !q$$

Both proofs may be given by tables over the four cases P (p) v resp. P (q) v = resp. ≠ ∅.

Both operators are not compatible with ′&′:

$\{p \,\&\, q\} \simeq \{p\} \,\&\, \{q\}$ $\quad !\,(p \,\&\, q) \simeq\ !p \mid !q$ are both not true.

This is due to the possible interference between *p* and *q* in the construct *p & q* by unification.

Concatenation ′,′ and insertion ′□′

$(p, ()) \simeq ((), p) \simeq p$

Proof:

$P\ (p, ())\ u = \bigcup_{v,\, w \text{ with } v \cdot w = u} P\ (p)\ v \oplus P\ (())\ w$
$= \bigcup_{v,\, w \text{ with } v \cdot w = u} P\ (p)\ v \oplus (\text{if } w = () \text{ then } E \text{ else } \varnothing)$
$= \bigcup_{v,\, w \text{ with } v \cdot w = u} (\text{if } w = () \text{ then } P\ (p)\ v \text{ else } \varnothing) = P\ (p)\ u$

The proof of the other statement is analogous.

Associativity: $(p, (q, r)) \simeq ((p, q), r)$ $\qquad$ $p \,\square_1\, (q \,\square_2\, r) \simeq (p \,\square_1\, q) \,\square_2\, r$

for all insertions $(\square_1, \square_2)$ that are associative on values with ′=′.
They are (Δ, Δ), (Δ, A), (A, A), (Λ, Δ), (Λ, A), (Λ, Λ).

Proof:

$P\ ((p, q), r)\ x$
$= \bigcup_{y,\, w \text{ with } y \cdot w = x} P\ (p, q)\ y \oplus P\ (r)\ w$
$= \bigcup_{y,\, w \text{ with } y \cdot w = x} (\bigcup_{u,\, v \text{ with } u \cdot v = y} P\ (p)\ u \oplus P\ (q)\ v\ ) \oplus P\ (r)\ w$
$= \bigcup_{y,\, w \text{ with } y \cdot w = x} \bigcup_{u,\, v \text{ with } u \cdot v = y} P\ (p)\ u \oplus P\ (q)\ v \oplus P\ (r)\ w$ by distributivity
$= \bigcup_{u,\, v,\, w \text{ with } (u \cdot v) \cdot w = x} P\ (p)\ u \oplus P\ (q)\ v \oplus P\ (r)\ w$
$= P\ (p, (q, r))\ x$ by the symmetry of the last term.

The proof of the associativity of $(\square_1, \square_2)$ is analogous.

If we have only ′≐′ or ′≑′, then the associativity for patterns does not hold since the great union is to be taken over different triples of values in this case.

Distributivity over ′|′:

$(p, (q \mid r)) \simeq (p, q) \mid (p, r)$ $\qquad$ $((p \mid q), r) \simeq (p, r) \mid (q, r)$
$p \,\square\, (q \mid r) \simeq (p \,\square\, q) \mid (p \,\square\, r)$ $\qquad$ $(p \mid q) \,\square\, r \simeq (p \,\square\, r) \mid (q \,\square\, r)$

Proof:

$P\ (p, (q \mid r))\ u$
$= \bigcup_{v,\, w \text{ with } v \cdot w = u} (P\ (p)\ v \oplus (P\ (q)\ w \cup P\ (r)\ w))$
$= \bigcup_{v,\, w \text{ with } v \cdot w = u} ((P\ (p)\ v \oplus P\ (q)\ w) \cup (P\ (p)\ v \oplus P\ (r)\ w))$ by distributivity
$= P\ (p, q)\ u \cup P\ (p, r)\ u = P\ ((p, q) \mid (p, r))\ u$

The other statements are analogous.

Distributivity over ′&′ is not true:

$(p, (q \,\&\, r)) \simeq (p, q) \,\&\, (p, r)$ is not true.
The other possible laws are also not true.

Example: $p = c?$ $\quad q = ()$ $\quad r = c$ $\quad$ application to value c.

$P\ (c?, (() \,\&\, c))\ c = (P\ (c?)\ () \oplus P\ (() \,\&\, c)\ c) \cup (P\ (c?)\ c \oplus P\ (() \,\&\, c)\ ())$
since $c = () \cdot c = c \cdot ()$
$P\ (c?)\ () = P\ (c?)\ c = E$ (later, we shall see that $p? \simeq () \mid p$)
$P\ (() \,\&\, c)\ c = \varnothing \oplus E = \varnothing$
$P\ (() \,\&\, c)\ () = E \oplus \varnothing = \varnothing$
thus $P\ (c?, (() \,\&\, c))\ c = (E \oplus \varnothing) \cup (E \oplus \varnothing) = \varnothing$
$P\ ((c?, ()) \,\&\, (c?, c))\ c = P\ (c?, ())\ c \oplus P\ (c?, c)\ c$
$P\ (c?, ())\ c = P\ (c?)\ c = E$ since $(p, ()) \simeq p$
$P\ (c?, c)\ c = (P\ (c?)\ () \oplus P\ (c)\ c) \cup (P\ (c?)\ c \oplus P\ (c)\ ())$

$= (E \oplus E) \cup (E \oplus \varnothing) = E$

together: $P\,((c?, ()) \,\&\, (c?, c))\; c = E \oplus E = E$

The essential of this example is that in *(c?, ()) & (c?, c)* applied to *c* the two occurrences of *c?* match differently: the first one matches value *c*, whereas the second one matches value *()*. In the other pattern *(c?, (c & ()))*, there is only one occurrence of *c?* that can only match one fixed value and thus, this pattern does not match value *c*. Note that this example does not bind names such that the law above cannot be proved by restriction of name usage.

Iteration

If $i < j$, then $p\,\{i..j\} \simeq p\,\{i\} \mid p\,\{i+1..j\}$

If $i < j < \infty$, then $p\,\{i..j\} \simeq p\,\{i..j-1\} \mid p\,\{j\}$

$p\,\{k+1\} \simeq (p, p\,\{k\})$ $\quad$ $p\,\{0\} \simeq ()$

$p\,\{1\} \simeq p$ $\quad$ $p\,\{2\} \simeq (p, p)$

$p? \simeq () \mid p$ $\quad$ $p* \simeq () \mid p+$

Proof: The properties in the first three lines follow directly from the definition of *P*.

$p\,\{1\} \simeq (p, p\,\{0\}) \simeq (p, ()) \simeq p$

$p\,\{2\} \simeq (p, p\,\{1\}) \simeq (p, p)$

$p? = p\,\{0..1\} \simeq p\,\{0\} \mid p\,\{1\} \simeq () \mid p$

$p* = p\,\{0..\infty\} \simeq p\,\{0\} \mid p\,\{1..\infty\} \simeq () \mid p+$

More properties of iterators ($i + \infty = i \cdot \infty = \infty$):

$(p\,\{i\}, p\,\{j\}) \simeq p\,\{i+j\}$ $\quad$ $(p\,\{i\})\,\{j\} \simeq p\,\{i \cdot j\}$

$(p\,\{i..j\}, p\,\{i'..j'\}) \simeq p\,\{i+i'..j+j'\}$

$p\,\{i..\} \simeq (p\,\{i\}, p*)$ $\quad$ $p+ \simeq (p, p*)$

Proof:

The first two properties may be proved by induction on *i*. The third statement is a little bit tricky, induction does not work since *j* and *j'* might be ∞. By definition of *P* for ',' and iterators, by using distributivity of ⊕ over union, and by the first statement one obtains

$$P\,(p\,\{i..j\}, p\,\{i'..j'\})\; v = \bigcup_{k=i}^{j} \bigcup_{l=i'}^{j'} P\,(p\,\{k+l\})\; v$$

Then one must verify that the sums k+l cover the whole range $\{i+i'..j+j'\}$.

$(p\,\{i\}, p*) = (p\,\{i..i\}, p\,\{0..\infty\}) \simeq p\,\{i+0..i+\infty\} = p\,\{i..\}$

$p+ = p\,\{1..\} \simeq (p\,\{1\}, p*) \simeq (p, p*)$

By the properties given so far, we see that we could have restricted the iterators to '*' without loss of computational power.

Other properties known from regular expressions may be claimed and proved.

In the next sections, we shall investigate the two aspects of the semantics of patterns - binding of names and defining a subset of matched values - separately. Between, we shall investigate the question of computability.

2.5. Name sets and normal patterns

We shall now consider the names bound by a pattern when applied to a value *v*. In the future transformation language there will be a construct '{p => e}' with pattern *p* and expression *e*, that denotes a mapping evaluating *e* in the environments produced by *p*. Then it is important to know whether all names occurring in *e* are really bound by *p*. In addition, the investigation of the sets of names bound by patterns will reveal more details about the semantics of patterns.

For patterns *p*, we shall define recursively the set *N(p)* of names that **may** be bound by *p*, and the set *n(p)* of names that are guaranteed to be bound. Then we shall prove that the recursively defined sets have really the property that we intuitively expect.

(2.5) Name sets N, n: $\mathrm{Pat} \to 2^{\mathrm{Name}}$

(1) $N\,(at) = \varnothing$ $\quad$ $n\,(at) = \varnothing$

(2)	N (op p)	= N (p)	n (op p)	= n (p)	
(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)	$= \varnothing$	n (!p)	$= \varnothing$	
(7)	N ({p})	$= \varnothing$	n ({p})	$= \varnothing$	
(8)	N (p, q)	$= N(p) \cup N(q)$	n (p, q)	$= n(p) \cup n(q)$	
(9)	N (p □ q)	$= N(p) \cup N(q)$	n (p □ q)	$= n(p) \cup n(q)$	
(10)	N (p {0..j})	= N (p)	n (p {0..j})	$= \varnothing$	
(11)	N (p {i..j})	= N (p)	n (p {i..j})	= n (p)	where i > 0

The definition of *N* is not surprising: a pattern may bind all names that may be bound by one of its direct constituents except *!p* and *{p}* that may not bind any name. Thus the set *N(p)* is simply the set of all names occurring in *p* except those under a not operator *!* or an unbind operator *{}*.

The definition of *n* is different from that of *N* only for alternations *p | q* and iterations starting from zero. When *p | q* is applied to a value, the resulting environments are produced by *p* or by *q* alone, and thus only the names that are bound by both *p* and *q* are guaranteed to be bound by *p | q*. If an iteration *p {0..j}* is applied to *()*, no names are bound by the match of *p {0}* to *()*, thus *n (p {0..j})* $= \varnothing$.

Proposition (2.6): For all patterns p, $n(p) \subseteq N(p)$ holds.

Proof: Straightforward induction on *p*.

The main property of *n* and *N* is formulated in the following theorem:

Theorem (2.7)

For all patterns p and all values v:
for all environments a in P (p) v, $n(p) \subseteq dom(a) \subseteq N(p)$.

Intuitively, this means that each environment produced by *p* binds at least the names in *n(p)* and at most the names in *N(p)*. The theorem neither claims that there is actually an environment *a* such that n(p) = dom (a) or dom (a) = N(p), nor that the intersection of all domains equals n(p) and the union equals N(p). Consider the pattern p = A | (B: c & d) with $n(p) = \varnothing$ and N(p) = {A, B}, but $P(p)\,v = \{[A \to v]\}$ for all values v.

Proof of the theorem by induction on p:

Let the first inclusion be (1) and the second one be (2).

Case p atomic or p = !q or p = {q} or p = q {0}:
P (p) v is either $\varnothing$ or E, thus dom (a) $= \varnothing$ for all a in P (p) v. Thus (2) is correct, and (1) is also correct since n(p) $= \varnothing$ in this case.

Case p = op q:
P (p) v is either $\varnothing$ or equals some P (q) v', and N(p) equals N(q), and n(p) n(q).

Case p = A (a name):
$P(p)\,v = \{[A \to v]\}$,
thus dom (a) = {A} = N(p) = n (p) for all a in P (p) v.

Case p = q & r or (q, r) or q □ r:
The theorem holds by induction hypothesis for q and r.
We have $N(p) = N(q) \cup N(r)$ and $n(p) = n(q) \cup n(r)$. In all three cases,
$P(p)\,v = \bigcup_i P(q)\,u_i \oplus P(r)\,w_i$ where u_i and w_i depend on v.
Let a be an arbitrary element of P (p) v. Then a is in $P(q)\,u_i \oplus P(r)\,w_i$ for some i.
Thus a = b + c where b in $P(q)\,u_i$ and c in $P(r)\,w_i$.
Hence $dom(a) = dom(b) \cup dom(c)$, and (1) and (2) hold by induction and monotony of $\cup$.

Case p = q {k} where k > 0:
Induction on k based on $q\{1\} \simeq q$ and $q\{k+1\} \simeq (q, q\{k\})$.

Case p = q {i..j}:

Then $P (p) v = \bigcup_{k=i}^{j} P (q \{k\}) v$ holds.

Take an arbitrary a in P (p) v. Then a is in P (q {k}) v for some k and thus $n (q \{k\}) \subseteq dom (a) \subseteq N (q \{k\})$.

Since N (q {k}) = N (q) = N (q {i..j}), we have (2).

If i = 0, we have $n (q \{i..j\}) = \emptyset$ and thus (1) holds.

Otherwise n (q {k}) = n (q) = n (q {i..j}) and we have (1), too.

Case p = q | r:

Here P (p) v is the union of P (q) v and P (r) v, and each element of P (p) v is in P (q) v or P (r) v. Let a be in P (q) v. Then

$n (q) \subseteq dom (a) \subseteq N (q)$ holds by induction hypothesis.

$n(p) = n(q) \cap n(r) \subseteq n(q) \subseteq dom(a) \subseteq N(q) \subseteq N(q) \cup N(r) = N(p)$.

Consider the construct ′{p => e}′ where *p* is a pattern and *e* an expression. It means that *e* is to be evaluated in the environments produced by *p* superposed with external environments. In order to assure that a name *A* occurring in *e* will be bound, we may check whether *A* is in *n(p)* or already bound outside of the construct.

Definition (2.8): A pattern p is normal iff n(p) = N(p).

Note that equivalence of a normal and a not normal pattern is possible, e.g. $() \mid (A: c \ \& \ d) \simeq ()$ where *c* and *d* are different nullary operators. If *p* and *q* are normal, then *p* | *q* need not necessarily to be, and a normal pattern may have subpatterns that are not normal.

There is a criterion for normality using only the mapping *N*. Unfortunately, it is only sufficing, not necessary i.e. there are normal patterns not satisfying the criterion.

Lemma (2.9)

(1) Atomic patterns, names, !p, and {p} are always normal.
(2) *op q* is normal iff *q* is.
(3) Patterns of shape p & q or (p, q) or p □ q are normal if p and q are normal.
The inverse direction is not true.
(4) p | q is normal iff p and q are normal and N(p) = N(q).
(5) p {i..j} with i > 0 is normal iff p is.
(6) p {0..j} is normal iff $N(p) = \emptyset$.

Proof:

(1) is true since $n(p) = N(p) = \emptyset$.

(2) n(op q) = n(q) and N(op q) = N(q).

(3) $n (p \ \& \ q) = n(p) \cup n(q)$ and $N (p \ \& \ q) = N (p) \cup N (q)$.

If p and q are both normal, the right hand sides are equal, and thus the left hand sides, too.

Example: ((A: c) | d) & (A: d) is normal, although (A: c) | d is not.

(4) $n (p \mid q) = n(p) \cap n(q)$ and $N (p \mid q) = N(p) \cup N(q)$.

If p and q are normal and N(p) = N(q) holds, we obtain n(p) = N(p) = N(q) = n(q) and thus n (p | q) = n(p) = N(p) = N (p | q).

Conversely, we have

$n(p) \cap n(q) \subseteq n(p) \subseteq N(p) \subseteq N(p) \cup N(q) = n(p) \cap n(q)$

and thus n(p) equals N(p). Analogously, we obtain n(q) = N(q).

N(p) = N(q) follows from $N(p) \cap N(q) = N(p) \cup N(q)$.

(5) n(p {i..j}) = n(p) and N(p {i..j}) = N(p) hold if i > 0.

(6) $n(p \{0..j\}) = \emptyset$ and N(p {0..j}) = N(p) hold.

Normality implies some pleasant properties:

Corollary (2.10)

1) If p is normal, then for all values v P (p) v is uniform.

2) If p is normal, then $p \ \& \ p \simeq p$.

Besides the idempotence, more algebraic laws may be proved if assumptions about the name sets are made.

Laws of absorption: Compare p & (p | q) and p | (p & q) with p.

Prop. (2.11)

1) If p is normal, then p & (p | q) $\simeq$ p | (p & q).
2) If N(q) $\subseteq$ n(p), then p | (p & q) $\simeq$ p.
3) If N(q) $\subseteq$ n(p) = N(p), then p & (p | q) $\simeq$ p.

Proof:

(3) is a direct consequence of (1) and (2).

(1) p & (p | q) $\simeq$ (p & p) | (p & q) $\simeq$ p | (p & q)

The first $\simeq$ holds by distributivity, the second one since p is normal (2.10).

(2) We must prove P (p) v $\cup$ (P (p) v $\oplus$ P (q) v) = P (p) v.

The direction $\supseteq$ is obvious, the other one follows from Lemma (1.14) since for all a in P (p) v and b in P (q) v holds dom(b) $\subseteq$ N(q) $\subseteq$ n(p) $\subseteq$ dom(a) and this implies P (p) v $\oplus$ P (q) v $\subseteq$ P (p) v.

The ′other′ distributivity is also valid under some assumptions:

Prop. (2.12)

If p is normal and N(q) and N(r) are subsets of N(p), then
p | (q & r) $\simeq$ (p | q) & (p | r)

Proof:

(p | q) & (p | r)	
$\simeq$ (p & p) | (p & r) | (q & p) | (q & r)	by distributivity
$\simeq$ p | (p & r) | (p & q) | (q & r)	since p is normal (2.10)
$\simeq$ p | (q & r)	by two absorptions (2.11)

2.6. The question of computability

Due to the property *p {i..}* $\simeq$ *(p {i}, p*)*, we need only prove the computability of the semantic function *P* for a starred pattern *p**.

Lemma (2.13)

For each pattern *p* and value *v*, there are numbers *c = c (p, v)* that only depend on *p* and *v* such that

P (p {k}) v = P (p {c}) v for all k $\geq$ c

$$P\,(p^*)\,v = \bigcup_{i=0}^{c} P\,(p\,\{i\})\,v$$

c may be UL(v) + max (1, |N (p) - n (p)|)

Proof:

The second statement follows directly from the first one since

$$P\,(p^*)\,v = \bigcup_{i=0}^{\infty} P\,(p\,\{i\})\,v.$$

Let v be a value with UL(v) = n, thus v = (v_1, ..., v_n) with atomic values v_i, and let k be a number with k $\geq$ n + 1.

p {k} is equivalent to (p, ..(k).., p) (where the notation *..(l)..* means *l times repeated*), and thus (by a computation like the one in the proof of associativity of ′,′)

$$P\,(p\,\{k\})\,v = \bigcup_{u_1,\dots,u_k:\ u_1\cdot\ldots\cdot u_k = v} P\,(p)\,u_1 \oplus ..(k).. \oplus P\,(p)\,u_k$$

Clearly, at most n of the k pieces u_1, ..., u_k of v may be different from (). Since concatenation with () is commutative and $\oplus$ is, too, we may collect the () pieces at the end:

$$P\,(p\,\{k\})\,v = \bigcup_{u_1,\dots,u_n:\ u_1\cdot\ldots\cdot u_n = v} P\,(p)\,u_1 \oplus ..(n).. \oplus P\,(p)\,u_n \oplus P(p)() \oplus ..(k-n).. \oplus P(p)()$$

This holds for all k $\geq$ n + 1. Let s be the set P (p) (), and let s^j = s $\oplus$ ..(j).. $\oplus$ s, then we must show that there are numbers i such that $s^j = s^i$ for all j $\geq$ i. Then c will be n+i.

For all environments a in s, n (p) $\subseteq$ dom (a) $\subseteq$ N (p) holds, and Lemma (1.18) implies our statement.

Here we see the danger of names under an iterator: if p does not bind any name, we have $|N(p) - n(p)| = 0$ and may choose $c(p, v) = UL(v) + 1$. Then the proof becomes easier, note that $P(p)()$ is either $\emptyset$ or E, and $\emptyset \oplus \emptyset = \emptyset$ and $E \oplus E = E$ hold.

Corollary (2.14)

The set of environments $P(p)\,v$ is finite for every pattern p and value v, and if the test *'v in L(at)'* for each atomic pattern *at* is computable, the semantic mapping P is computable, too.

2.7. Language of a pattern, linear patterns and types

First, we shall introduce some more classes of patterns.

Definition (2.15)

A pattern is called
- a free type, if it does not contain any name.
- type, if it does not bind any name, i.e. N(p) = $\emptyset$.
- a linear type, if it is both a type and a linear pattern.

A pattern is linear iff
- for all its subpatterns of shape p & q or (p, q) or (p $\square$ q), N(p) and N(q) are disjoint
- and for all its subpatterns of shape p {i..j}, N(p) is $\emptyset$, i.e. p is a type.

Examples: *c & d* is a free type, *{add (E, E)}* a type, but not a linear type, and *((A: c), !(A: c))* is linear, but *((A: c), (A: c))* is not.

A free type is linear since for all its subpatterns p, N(p) is $\emptyset$.

A type is a normal pattern since N(p) = $\emptyset$ implies n(p) = N(p) = $\emptyset$ by Prop. (2.6).

Thus we have the following hierarchy:

$$\text{Free types} \subseteq \text{linear types} \begin{array}{l} \subseteq \text{types} \subseteq \text{normal patterns} \subseteq \\ \subseteq \text{linear patterns} \subseteq \end{array} \text{patterns}$$

Each subpattern of a free type is again a free type, and each subpattern of a linear pattern is again linear. The other classes do not share this property:

{A: c} is a linear type, but A: c is not a type.
{(A: c) | d} is normal, but (A: c) | d is not.

Each pattern p may be transformed into a type by the unbind operator *{}*. Further, we define a mapping F deleting all names out of a pattern and thus transforming a pattern into a free type.
It may recursively defined by equations such as
F(at) = at F(A) = all F({p}) = {F(p)} F (p & q) = F(p) & F(q) etc.

For a type t and a value v, $P(t)\,v$ is either $\emptyset$ or E, thus the semantics of a type is essentially to designate the subset of values where $P(t)\,v$ is not empty. In this case, we say that these values are those of type t. For an arbitrary pattern, we call the set of values where $P(p)\,v$ is not empty the language matched by the pattern.

Definition (2.16)

Language described by a pattern:
L: Pat $\rightarrow 2^{\mathrm{Val}}$, L(p) = { v in Val | P (p) v $\neq \emptyset$ }

The language described by a pattern cannot be directly denotationally defined, and equality of languages is not a congruence relation.

Example:

Let *op* be an operator and *at* an atomic pattern.

op (A: at) and *op (B: at)* describe the same language, but
A: op (A: at) and *A: op (B: at)* do not since the first pattern describes the empty language due to unification.

But for types t1 and t2 holds

(2.17) if L(t1) = L(t2) then t1 ≃ t2

since P (t) v = (if v in L(t) then E else ∅) for types t.

Now we investigate how the language of a pattern depends on the languages of its constituents. For languages, we introduce some operations besides of union and intersection.

Definition (2.18)

Let L and M be subsets of Val.

L · M	= { u · v | u in L and v in M }
L □ M	= { u □ v | u in L and v in M and u □ v is defined }
L ˆ n	= { v ·...· v (n times) | v in L }
L^n	= L ·...· L (n times)
op L	= { op v | v in L } for operator op
L - M	= { v | v in L, but not in M }

Theorem (2.19)

L (at)	= L (at)	(see below)	
L (A)	= Val	L (op p)	= op L(p)
L ({p})	= L(p)	L (!p)	= Val - L(p)
L (p | q)	= L(p) ∪ L(q)	L (p & q)	⊆ L(p) ∩ L(q)
L (p, q)	⊆ L(p) · L(q)	L (p □ q)	⊆ L(p) □ L(q)

If N(p) ∩ N(q) = ∅, then

		L (p & q)	= L(p) ∩ L(q)
L (p, q)	= L(p) · L(q)	L (p □ q)	= L(p) □ L(q)

L (p {i..j}) $= \bigcup_{k=i}^{j}$ L (p {k})

L (p {0}) = {()}   L(p) ˆ k ⊆ L (p {k}) ⊆ $L(p)^k$

If N(p) = ∅, then L (p {k}) = $L(p)^k$

L (at) = L (at) means that the language of the pattern *at* as defined in (2.16) equals the predefined language of the atomic pattern *at* as introduced at the beginning of section (2.3).

Proof:

The first five statements are obvious.
v in L (p | q) iff P (p | q) v ≠ ∅ iff P (p) v ∪ P (q) v ≠ ∅
iff P (p) v ≠ ∅ or P (q) v ≠ ∅ iff v in L(p) or v in L(q) iff v in L(p) ∪ L(q)
If v is in L (p & q), then P (p) v ⊕ P (q) v is not ∅.
Thus P (p) v and P (q) v are not empty, therefore v is in L(p) and in L(q),
hence v is in the intersection of L(p) and L(q).
v in L (p, q) implies $\bigcup_{u,\, w \text{ with } u \cdot w = v}$ P (p) u ⊕ P (q) w ≠ ∅
Thus there are u, w such that u · w = v and P (p) u ⊕ P (q) w ≠ ∅
Hence P (p) u and P (q) w are not empty, therefore
u is in L(p) and w in L(q) and finally v = u · w in L(p) · L(q).
The statement about □ is analogous.
From the three equalities holding if N(p) and N(q) are disjoint,
we consider only the second one:
Let v be in L(p) · L(q). Then there are values u, w such that
v = u · w and u in L(p) and w in L(q).
Hence there are environments a in P (p) u and b in P (q) w.
Since N(p) and N(q) are disjoint, dom (a) and dom (b) are disjoint, too,

and thus a + b is in P (p) u ⊕ P (q) w, and thus v in L (p, q).
p {i..j}: similar to p | q.
Relations between L (p {k}) and $L(p)^k$ may be proved by induction.
The condition N(p) = ∅ stems from N(p) ∩ N(p {k}) = ∅
when considering (p, p {k}).
Let v be in L(p) ˆ k. Then $v = w^k$ where w is in L(p).
Thus there is an environment a in P (p) w.
Then a = a + ... + a ∈ P (p) w ⊕...⊕ P (p) w ⊆ P (p) v.

Now we can state the key property of linear patterns:

Theorem (2.20)

The language of a pattern is contained in the language of the corresponding free type: L(p) ⊆ L(F(p)).
The names in a linear pattern do not affect the matched language: if p is linear, then L(p) = L(F(p)).

Proof:

By induction on p using the previous theorem. Note that the precondition of linearity just implies the equality statements of the theorem.

Consequences:

1) The unbind operator is not needed for linear patterns:
If p is linear, then {p} ≃ F(p).
2) Names under negation may be omitted in a linear pattern:
If p is linear, then !p ≃ ! F(p)
3) Linear types may be replaced by free types.
If p is linear type, then p ≃ F(p).

Proof:

1) Since p is linear, we have L({p}) = L(p) = L(F(p)), and since {p} and F(p) are both types, the equality of the language implies equivalence (2.17).
2) !p ≃ ! {p} ≃ ! F(p)
3) Since p is linear, we have L(p) = L(F(p)), and since both p and F(p) are types, this implies their equivalence.

Free types satisfy all preconditions that we needed for the proof of the laws of lattice theory, in addition we have
!!p ≃ {p} ≃ p and
! (p & q) ≃ ! (!!p & !!q) ≃ !! (!p | !q) ≃ !p | !q
for free types, and we obtain

Corollary (2.21)

The set of free types with the operations *&*, | and *!* is a boolean algebra modulo semantic equivalence that is closed under all syntactic operations on patterns. The mapping *L* is a homomorphism from this algebra to the powerset of *Val*.

2.8. Degeneration of patterns

A pattern *p* is called degenerated if its language is empty. Two kinds of degeneration may be distinguished: degeneration due to unification, then *L(p)* is empty, but the language of the corresponding free type *L(F(p))* is not; and degeneration due to structural incompatibility, then both *L(p)* and *L(F(p))* are empty.

Degeneration is said to be caused by a pattern, if its language is empty but the languages of its direct constituents are not. Looking at theorem (2.19), one can detect the shapes of patterns that may cause degeneration.

The influence of unification is important in those cases where theorem (2.19) claims only $\subseteq$, not equality.

Examples: Let *c* and *d* be distinct constants.

We give an example for a pattern *p* causing degeneration, and a pattern *q* equivalent to *F(p)* to show that the degeneration is due to unification.

And operator:	add ((A: c), d) & add (c, (A: d))	add (c, d)
Comma operator:	(A: c, A: d)	(c, d)
Insertion:	add (A: c, []) Λ (A: d)	add (c, d)

Iteration may not cause degeneration since $L(p) \neq \emptyset$ implies $L(p) \hat{} k \neq \emptyset$ for all k and thus L (p {i..j}) $\neq \emptyset$.

Reasons for structural incompatibility may be detected by looking at the operations combining the languages of the constituents of linear patterns to the language of the whole pattern. Union and product ′ · ′ of non-empty languages are again non-empty.

!p may cause degeneration if L(p) = Val, e.g. ! (c | !c).

p & q if the languages of *p* and *q* are disjoint e.g. c & d

p □ q if the values matched by *p* and *q* do not fit together such that their insertion is never defined e.g. c Λ c since LL(c) = 0.

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]