

A Study of General Tracer (I) --denotational description of Prolog--

S. OHYAGI

In this paper a state transition model of pure Prolog with cut control primitive is presented in which each program construct is interpreted as a continuous Lambda function. This description has two main purposes. One is to simplify stack information so that a state includes only the information of stack frame plus two control statuses. The other is to preserve set function interpretation of prolog program so that the retract of fixed point set has logical meaning. Continuation model is not presented. It is thought to be unnecessary because Prolog has no "go to" construct.

§1 Introduction

There are several papers on the semantics of Prolog. Pioneering works on this subject are Van Emden, Kowalski¹⁾ and Apt, Van Emden²⁾, which discuss the fixed point semantics of Prolog as a Horn sentence. Lasse-Maher³⁾ proposed to interpret Prolog program as a monotone increasing function on the Plotkin's T^ω domain. In this paper Prolog program is interpreted as a continuous function on the retract E of T^ω . The advantage of using E as a data domain is that the determination of failure becomes continuous on this domain.

One of the key point of semantical description is to give meaning to each program construct. Especially in the denotational description the meaning to be given is a continuous function. This principle is adopted in this paper. The other object of the semantical description in this paper is to simplify state description. The usual Prolog interpreter uses stack. However very limited part of the stack is necessary for the program construct. So stack information should be abstracted if it is used as a state. The state adopted in this paper includes the information on one stack frame with some control parameters. One of the control parameters is success or failure of the process. The other one is the number of success passes of the process. The adoption of this state description gives a very different appearance

to the description of the semantics compared to the usual interpreter. However it offers clear and simple meaning to every program construct.

Finally definability of Prolog program on the real data domain is considered. The real data domain is a subspace E_0 of E but Prolog program is not a totally defined function on E . It is shown that Prolog program is defined on $E_0 \setminus A$ for some subset A where the closure of A does not include any open set.

§2 The domain of Prolog programs

In this paper a clause of a Prolog program is interpreted as a set function. This is the same idea of Kowalski and Emden. This treatment is important if you want to explain the phenomena, variable passing, which is peculiar to Prolog. Further if you treat undefined variable as only symbols then logical meaning of the program will be lost. This section discusses the domain of Prolog programs and propose a treatment of clauses as continuous functions.

The set FT of finite trees is defined inductively as follows.

[Definition 1]

Let $\{c_n \mid n \geq 1\}$ be a set of constant symbols and let $\{f_m^n(x_1, \dots, x_m) \mid 1 \leq n, m \leq N\}$ be a set of function symbols. The FT is defined as a minimum set satisfying

the following conditions.

i) $c_n \in FT$ where $n \geq 1$

ii) If $\alpha_1, \dots, \alpha_m \in FT$ then

$$f_m^n(\alpha_1, \dots, \alpha_m) \in FT$$

The set FT plays a role of Herbrand universe in first order logic. We denote the power set of FT with Scott topology by $P(FT)$. Formally this is defined as follows.

[Definition 2]

Let α be a finite subset of FT. The set $\text{above}(\alpha)$ is defined as

$$\text{above}(\alpha) = \{x \mid x \supset \alpha, x \subset FT\}$$

Then T the base of the topology of $P(FT)$ is defined as

$$T = \{\text{above}(\alpha) \mid \alpha \text{ is a finite subset of FT}\}.$$

On the space $P(FT)$ the function $\|p(x) \leq q(x)\|$ defined as

$$\|p(x) \leq q(x)\|(u) = \{q(y) \mid p(y) \in u\}$$

is a continuous function.

However the function $\|\forall x \neg p(x) \leq z\|$ defined as

$$\|\forall x \neg p(x) \leq z\|(u) = \begin{cases} z & \text{in case } p(x) \notin u \text{ for all } \\ & x \in FT \\ \emptyset & \text{else} \end{cases}$$

is't even monotone increasing. Prolog interpreter needs such an operation when it selects next alternative clause after the current unification failed.

Let us take the domain T as

$$T = \{\langle u, v \rangle \mid u, v \in P(FT), u \cap v = \emptyset\}.$$

The space T is essentially Plotkin's T^ω and this definition is similar to that of Barendregt. The first element u of the pair $\langle u, v \rangle$ is seen to be positive information and the second v to be negative information. Then $\|p(x) \leq q(x)\|$, $\|\forall x \neg p(x) \leq z\|$ are redefined as a function on T in the following.

$$\begin{aligned} \|p(x) \leq q(x)\|(\langle u, v \rangle) &= \langle \{q(y) \mid p(y) \in u\}, (P(FT) \setminus \{q(y) \mid y \in FT\}) \\ &\quad \cup \{q(y) \mid p(y) \in v\} \rangle \end{aligned}$$

$$\begin{aligned} \|\forall x \neg p(x) \leq z\|(\langle u, v \rangle) &= \begin{cases} z & \text{in case } v \supset \{p(x) \mid x \in FT\} \\ \emptyset & \text{else} \end{cases} \end{aligned}$$

Then the function $\|p(x) \leq q(x)\|$ is continuous and the function $\|\forall x \neg p(x) \leq z\|$ is monotone increasing. This is the J-L Lassey and M.J. Maher's approach. The main point of this chapter is the function $\|\forall x \neg p(x) \leq z\|$ can be made into a continuous function on certain

subspace of T with somewhat different topology.

[Definition 3]

Let F be a finite tree of FT with constant symbols c_{n_1}, \dots, c_{n_r} ($n_1, \dots, n_r > N$) on its leaves. If it has constant symbol c_k on its leaf and c_k satisfies

$$k \neq n_1, \dots, n_r \Rightarrow k \leq N$$

then we write down F as

$$F = F(c_{n_1}, \dots, c_{n_r}).$$

Define function b and \bar{b} as

$$\begin{aligned} b : FT \rightarrow P(FT) ; b(F(c_{n_1}, \dots, c_{n_r})) \\ = \{ F(x_1, \dots, x_r) \mid x_1, \dots, x_r \in FT \} \end{aligned}$$

$$\bar{b} : P(FT) \rightarrow P(FT) ; \bar{b}(x) = \bigcup_{F \in x} b(F)$$

[Proposition 1]

$\bar{b} : P(FT) \rightarrow P(FT)$ is a continuous retraction map.

(Proof)

omitted.

[Definition 4]

Let $FT(N)$ be the set of finite trees which has constants with lower case indexes all less than or equal to the number N on their leaves. The power set of $FT(N)$ is denoted as $P(FT(N))$ which is assumed to have subspace topology of $P(FT)$.

Our concern is to obtain a retract of $P(FT(N)) \times \bar{b}(P(FT))$ as a similar correspondent of T^ω .

[Proposition 2]

Let r be a map

$$\begin{aligned} r : P(FT(N)) \times \bar{b}(P(FT)) \rightarrow P(FT(N)) \times \bar{b}(P(FT)) \\ ; r(\langle x, y \rangle) = \begin{cases} \langle x, y \rangle & \text{in case } x \cap y = \emptyset \\ \top & \text{else} \end{cases} \end{aligned}$$

The map r is a continuous retraction map.

(Proof)

omitted.

Please refer to Scott [5] as to retraction maps.

[Definition 5]

The space E is defined as a retract using a retraction map r. That is

$$E = r(P(FT(N)) \times \bar{b}(P(FT))).$$

E is a continuous lattice and so a retract of $P(\omega)$. On the space $\bar{b}(P(FT))$ which represents negative information of the space E $\text{above}(\bar{b}(F))$ defined as

$$\text{above}(\bar{b}(F)) = \{x \mid x \in \bar{b}(P(FT)), \bar{b}(F) \subset x\}$$

is open.

While

$$\begin{aligned} & \| \forall x \neg p(x) \leq z \| (\langle u, v \rangle) \\ &= \begin{cases} z & \text{in case } \forall x \in FT, p(x) \in v \\ \emptyset & \text{else} \end{cases} \\ &= \begin{cases} z & \text{in case } \bar{b} (p(c_{n+1})) \subset v \\ \emptyset & \text{else} \end{cases} \\ &= \begin{cases} z & \text{in case } v \in \text{above} (\bar{b} (p(c_{n+1}))) \\ \emptyset & \text{else} \end{cases} \end{aligned}$$

holds.

So $\| \forall x \neg p(x) \leq z \|$ is continuous on E. The function $\| p(x) \leq q(x) \|$ on E defined as

$$\begin{aligned} & \| p(x) \leq q(x) \| (\langle u, v \rangle) \\ &= \langle \{ q(y) \mid p(y) \in u, y \in FT(N) \}, \\ & \quad (P(FT) \setminus \{ q(y) \mid y \in FT \}) \\ & \quad \cup \{ q(y) \mid p(y) \in v, y \in FT \} \rangle \end{aligned}$$

also is continuous on E.

Let us explain Prolog in this paper and how it is seen as a function on the space E. The syntax of Prolog subset is defined as follows.

$$\begin{aligned} \text{Program} &::= \text{Clause_groups} \\ \text{Clause_groups} &::= \text{Clause_group} \mid \\ & \quad \text{Clause_group Clause_groups} \\ \text{Clause_group} &::= \text{Clauses} \\ \text{Clauses} &::= \text{Clause} \mid \text{Clause clauses} \\ \text{Clause} &::= \text{Clause_head} :- \text{And_part} \\ \text{Clause_head} &::= \text{Predicate_functor} \\ \text{And_part} &::= \text{terms} \\ \text{terms} &::= \Lambda \mid \text{terms term} \mid \text{terms} ! \\ \text{term} &::= \text{Predicate_functor} \\ \text{Predicate_functor} &::= f_j^i(\overbrace{\text{argument}, \dots, \text{argument}}^j) \\ \text{argument} &::= x_i \mid f_j^i(x_{i_1}, \dots, x_{i_j}) \mid c_1 \mid \dots \mid c_N \mid \\ & \quad f_j^i(\overbrace{\text{argument}, \dots, \text{argument}}^j) \end{aligned}$$

For example

$$\begin{aligned} p(x) &:- q(f(x)), r(x,y) \\ p(g(x)) &:- r(x,z) \\ q(f(g(x))) &:- \\ r(u(x),z) &:- \end{aligned}$$

where p,q,r,f,g are function symbols of $\{ f_j^i \mid 1 \leq i, j \leq N \}$

is a Prolog program. If you ask

$$p(x)?$$

to this program then it returns

$$x = g(u(y)).$$

If you insert ! after q(f(x)) in the above program then

we obtain the following program.

$$\begin{aligned} p(x) &:- q(f(x)) ! r(x,y) \\ p(g(x)) &:- r(x,z) \\ q(f(g(x))) &:- \\ r(u(x),z) &:- \end{aligned}$$

If you ask the same question p(x)? then the interpreter first select the top most clause of the list and matches the clause head and the current term p(x). The interpreter proceeds to each And_part terms and finally at the last term of predicate name r it becomes a question r(g(x),y)?. But it fails. So it will go backtrack. However the backtrack is prevented by the cut symbol ! and the entire clause group of the predicate name p(x) fails. This cause total failuar and the interpreter returns "fail". To see program as a function on the space E the question p(x)? is identified the element α of E as

$$\alpha = \langle \{ p(x) \mid x \in FT \}, \bar{b}(FT) \setminus \{ p(x) \mid x \in FT \} \rangle$$

and the answer $x = g(u(y))$ is identified with the element β of E as

$$\beta = \langle \{ p(g(u(y))) \mid y \in FT(N) \}, \bar{b}(FT) \setminus \{ p(g(u(y))) \mid y \in FT \} \rangle$$

in the first program.

So that program maps α to β as a function on E. The second elements of both α and β are roughly the compliments of the first elements of them. This means α and β represent complete information. So the data domain of Prolog in ordinary sense is the subspace E_0 of E defined as follows.

[Definition 6]

Let E_0 be

$E_0 = \{ \langle \alpha, \beta \rangle \mid \langle \alpha, \beta \rangle \in E \text{ and } \langle \alpha, \beta \rangle \text{ is a maxima element of } E \setminus \{ \top \} \text{ as to the order derived from the lattice structure of } E \}$

The space E_0 is provided to include undefined elements such as infinite loops.

The space E_0 is somewhat complicated. To give clear view let us transform E_0 to simpler space.

[Definition 7]

Let α be a finite subset of FT(N) and β be a finite subset of FT.

Define $U(\alpha, \bar{b}(\beta))$ as

$$U(\alpha, \bar{b}(\beta)) = \{ x \mid x \subset FT(N), x \supset \alpha \text{ and } x \subset FT(N) \setminus \bar{b}(\beta) \}.$$

The space E_0^* is defined as a space $P(FT(N))$ with

topology generated by T defined as

$$T = \{ U(\alpha, \bar{b}(\beta)) \mid \alpha \in FT(N), \beta \in FT \text{ and } \alpha, \beta \text{ are finite or empty} \}$$

Then the following proposition holds.

[Proposition 3]

E_0 and E_0^* are homeomorphic.

(Proof)

It is sufficient to show that the map I

$$I : E_0 \rightarrow E_0^* ; \langle x, y \rangle \rightarrow x$$

is a homeomorphism. I is one to one and onto. Let us prove that for the open set $V(\alpha, \bar{b}(\beta))$ of E_0 defined as

$$V(\alpha, \bar{b}(\beta)) = \{ \langle u, v \rangle \mid u \in \text{above}(\alpha), v \in \text{above}(\bar{b}(\beta)) \}$$

$$I(V(\alpha, \bar{b}(\beta))) = U(\alpha, \bar{b}(\beta))$$

holds.

Let $\langle u, v \rangle$ be an element of $V(\alpha, \bar{b}(\beta))$.

Then $v' = v \cap FT(N)$

satisfies

$$v' \in \text{above}(\bar{b}(\beta) \cap FT(N))$$

and $u \cap v' \subset u \cap v = \emptyset$.

So

$$FT(N) \setminus u \supset v' \supset \bar{b}(\beta) \cap FT(N)$$

holds.

That is $u \subset FT(N) \setminus (\bar{b}(\beta) \cap FT(N))$.

$$\therefore u \in U(\alpha, \bar{b}(\beta)).$$

This shows

$$I(V(\alpha, \bar{b}(\beta))) \subset U(\alpha, \bar{b}(\beta)).$$

Conversely if $u \in U(\alpha, \bar{b}(\beta))$ then

$$\alpha \subset u \text{ and } u \subset FT(N) \setminus (\bar{b}(\beta) \cap FT(N)).$$

So

$$u \cap \bar{b}(\beta) \cap FT(N) = \emptyset.$$

From $u \subset FT(N)$

$$u \cap (\bar{b}(\beta) \setminus FT(N)) = \emptyset.$$

This means

$$u \cap \bar{b}(\beta) = \emptyset.$$

$\langle u, v \rangle = I^{-1}(u)$ is a maximal element of $E \setminus \{ \top \}$.

So

$$\langle u, \bar{b}(\beta) \rangle \sqsubset \langle u, v \rangle.$$

$$\therefore v \supset \bar{b}(\beta).$$

This means

$$u \in \text{above}(\alpha), v \in \text{above}(\beta).$$

$$I^{-1}(u) \in V(\alpha, \bar{b}(\beta)).$$

I is one to one and onto continuous open mapping. So

I is a homeomorphism.

Q.E.D.

This proposition says that Prolog program on the space E_0 can be viewed as a function on the space E_0^* . The program already stated in the example maps $\{ p(x) \mid x \in FT(N) \}$ to $\{ p(g(u(y))) \mid y \in FT(N) \}$ as a function on the space E_0^* .

§2 A state transition model of Prolog

To describe the function of language interpreter compactly and without entering into full details of practical implementation technique is one of the important point of denotational semantics. To localize the state information for the current program construct to be described and to take off all unnecessary stack information is a key to the success of the denotational description of Scott, Strachy, Miler, Stoy and others.

The theme of this section is to apply the same paradigm to the description of Prolog. Here some reference should be made to the Prolog interpreter. The data structure of the stack for our Prolog interpreter is as follows.

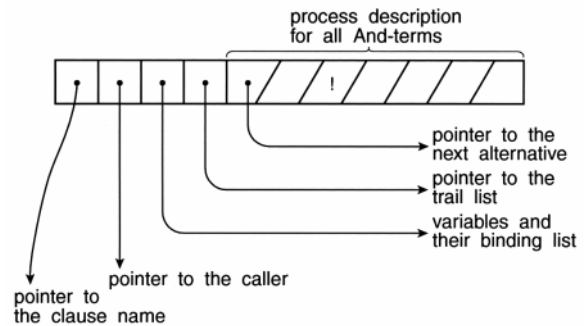


Fig.1 The data structure of the stack for our prolog interpreter

For example the stack frame created for the clause $p(x) :- q(x,y) ! r(y,x)$ when it is applied to the question $p(a)?$ is as in the following.

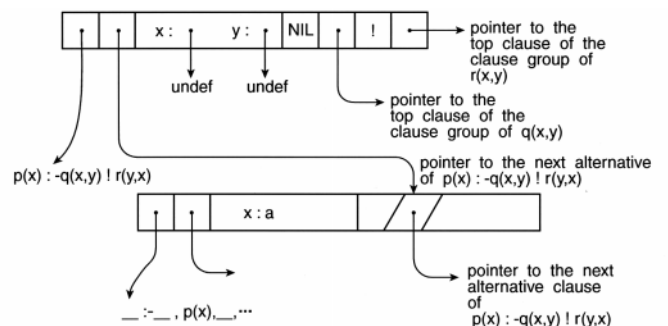


Fig.2 The stack frame created for the clause $p(x) :- q(x,Y) ! r(y,x)$.

After the clause is applied, $p(a)$ and the head $p(x)$ of the clause are unified and x is bound to a . Let us try an example.

If you ask $p(x)?$ to the Prolog program

```
p(x) :- q(f(x)) ! r(x,y)
p(g(x)) :- r(x,z)
q(f(g(x))) :-
r(u(x),z) :-
```

The interpreter creates

- 1) the stack frame for the clause $:- p(x)$.

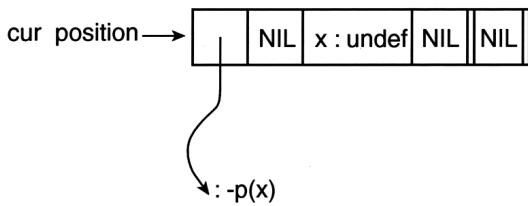


Fig.3.1 The data structure of the stack for our prolog interpreter

Then it

- 2) unifies the term of the clause $:- p(x)$ and the head of the clause $p(x) :- q(f(x)) ! r(x,y)$ and push the frame of the latter clause on the stack.

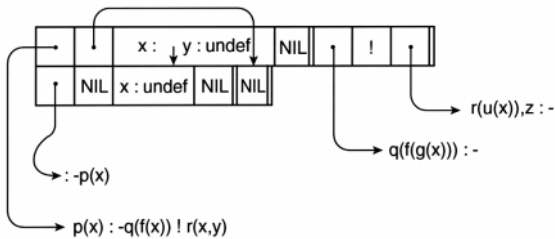


Fig. 3.2 The stack frame created for the clause $p(x):-q(x,y) ! r(y,x)$

Then it

- 3) proceeds to the first term of `And_part`, unifies $q(f(x))$ and $q(f(g(x)))$ and push the frame of the clause $q(f(g(x))) :-$ to the stack. At the same time it saves the binding information, which cannot be taken away by simple pop of the stack, in the trail stack. This is important in the case of backtrack.

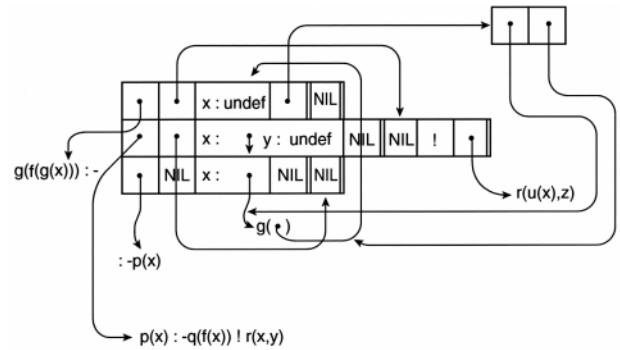


Fig.3.3 unification of $q(f(x))$ and $q(f(g(x)))$ and push the frame of the clause $q(f(g(x))) :-$ to the stack

The interpreter thus proceeds.

It will be reasonable to take one frame of the stack as a state. So the data structure of a state is defined as follows.

```
state = < clause_name, variable_assignment,
list_of_partial_state, process_status >
```

The structure sharing technique is adopted in the above hand simulated interpreter. However structure sharing causes simultaneous bindings of all the parameters which share the variable to be bound. It is very difficult to reflect such a mechanism to state transition model. And it is also unnecessary for logical description of Prolog function. So simple principle of copying is adopted. Thus trail list becomes unnecessary. Bindings of variables change at each step after the execution of each term of `And_part`. So it is necessary to retain variable assignment in the `partial_state` after the execution of each term of `And_part`.

After the execution of each term of `And_part`, the hand simulated interpreter stated above must have pushed stack frames corresponding to the execution from the first term to the current term. Such execution sequence may include many backtracks. So it is desirable to retain the number of success passes in the `partial_state` so as to determine the execution sequence uniquely. So the data structure of `partial_state` is as follows.

```
partial_state= <next_alternative_clause,
variable_assignment,backtrack_state>
backtrack_state = <number_of_success_passes,
number_of_the_most_recent_success_
passes>|<|>
```

Finally the indicator of success or failuar of the process when it reached current state and the index which

indicate current_And_term are required. This is called process_status.

process_status = <{suc,fail,! fail},index>

Let us start the semantical description of Prolog using state transition model. The data type of state is written as S. According to Scott [5] data type is a retract. So S is taken to be a retract of P(ω). The flat lattice of natural numbers is written as $\hat{\omega}$. The function m(s) assigns meanings to the program constructs of Prolog. And we would like to define

m(s)[[Program]]: $\hat{\omega} \times E \rightarrow E$
m(s)[[Clause_groups]],
m(s)[[Clause_group]],
m(s)[[Clauses]],
m(s)[[Clause]],
m(s)[[And_part]],
m(s)[[terms | terms]],
m(t)[[term | terms]]: $\hat{\omega} \times S \rightarrow S$

and tran(terms): S \rightarrow S.

m(s)[[Program]](n)(,v>)
= m(s)[[Clause_groups]](n)(,v>)
m(s)[[Clause_group Clause_groups]](n)(,v>)
= logical_part(m(s)[[Clause_group]](n)(set_
flame(F(Clause_group)(,v>), Clause_
group_name)))
 $\dot{\cup}$ m(s)[[Clause_groups]](n)(,v>)

Here F(Clause_group) denotes a continuous function on E_0 defined as

F(c_i) = $\| c_i \leq c_i \|$
F(f_j^i) = $\| f_j^i(x_1, \dots, x_j) \leq f_j^i(x_1, \dots, x_j) \|$

according to the Clause_group name of c_i and f_j^i

set_flame(α, β)
= < :- $\beta, \alpha, <NIL, NIL, <0, 0>>, <suc, 0>>$
logical_part(<p:- $q_1, \dots, q_r, \alpha, \beta, <suc, k>>$)
= $\| M_j(x_1, \dots, x_j) \leq q_r \|(\alpha)$
logical_part(<p:- $q_1, \dots, q_r, \alpha, \beta, <fail, k>>$)
= {fail}

$\langle u_1, v_1 \rangle \dot{\cup} \langle u_2, v_2 \rangle = \langle u_1 \cup u_2, v_1 \cap v_2 \rangle$

m(s)[[Program]](n)(,v>) applies Prolog program for the input <u,v>, an element of E, and returns output after n times of success passes. This is the same for m(s)[[Clause_groups]]. The function m(s)[[Clause_groups]], m(s)[[Clause_group]], and m(s)[[Clause_group Clause_groups]] using the syntax definition of Clause_group ::= Clause_group Clause_groups |

Clause_group. Since m(s)[[Clause_group]], is a function from $\hat{\omega} \times S$ to S initial state should be established by applying set_flame to the part of <u,v> which corresponds to the current Clause_group_name. This part of <u,v> is obtained by applying function F(Clause_group). Further $\dot{\cup}$ is used to sum up all the result of application of m(s)[[Clause_group]](s). For predicate_functors p(x,y) and q(x,y) the function $\| p(x,y) \leq q(x,y) \|$ on E is defined as

$\| p(x,y) \leq q(x,z) \|(\langle u,v \rangle)$
= <{ $q(x,z) \mid \exists y p(x,y) \in u, z \in FT(N)$ },
 $(FT \setminus \{ q(x,z) \mid x,z \in FT \}) \cup \{ q(x,z) \mid$
 $\forall y p(x,y) \in v, z \in FT(N) \}$ >

and it is continuous.

The function $\| M_j(x_1, \dots, x_j) \leq q_k \|$ is a function in this sense and this notation shall be used hereafter.

m(s)[[Clause_group]](n)(s)
= m(s)[[Clauses]](n)(s)

This shows the meaning of Clause_group is the total meaning of all the component Clauses.

m(s)[[Clause Clauses]](n)(s)
= if Fail(head(Clause))(cur(s))
then m(s)[[Clauses]](n)(next(s)),
if Suc(head(Clause))(cur(s))
then
if n \leq n(Clause)(s)
then
m(s)[[Clause]](n)(s)
else
if Clause_exhausted(
m(s)[[Clause]](n)(s))
then
m(s)[[Clause]](n)(s)
else
m(s)[[Clauses]](n-n(Clause(s)))(
next_clause(s))

Here

cur(<p:- $q_1, \dots, q_r, \alpha, <a_1, \dots, a_{k-1}, <cl,$
,v>,bt>, ...>, <c,k>>) = $\| M_j(x_1, \dots, x_j) \leq$
 $q_{k+1} \|(\langle u,v \rangle)$

j is the number of free variables which appear in the clause p:- q_1, \dots, q_r and M_j is a fixed predicate of j variables.

Fail(q)(,v>) = $\| \forall x \neg q \leq 1 \|(\langle u,v \rangle)$

Suc(q)(,v>) = $\| q \leq 1 \|(\langle u,v \rangle)$

next(<cl, $\alpha, <a_1, \dots, a_k, <cl', \gamma, bt>, \dots, <c,k>>$)

$= \langle cl, \alpha, \langle a_1, \dots, a_k, \langle cl' + 1, \dots, \dots, \langle c, k \rangle \rangle \rangle$
 Clause_exhausted($\langle p: -q_1, \dots, q_r, \alpha, \langle a_1, \dots, a_k, \langle cl, \gamma, bt \rangle, \dots, \langle c, k \rangle \rangle \rangle$)

$= \begin{cases} 1 & \text{in case } cl \text{ exceeds the number of all the} \\ & \text{clauses in the clause group of } q_k \\ 0 & \text{else} \end{cases}$

length_match(Clause_list, s)

$= \begin{cases} 1 & \text{in case } s = \langle p: -q_1, \dots, q_r, \alpha, \langle a_1, \dots, a_k, \langle cl, \langle u, v \rangle, bt \rangle, \dots, \langle c, k \rangle \rangle \rangle \text{ and the number} \\ & \text{of causes in the clause_group of} \\ & q_{k+1} - cl + 1 = \text{the length of the Clause_} \\ & \text{list holds} \\ 0 & \text{else} \end{cases}$

n(Clause)(s) and next_clause(s) shall be explained later.

In case that there is no clause corresponding to the current clause number in the partial state column of the state s the current state does not move.

In case current clause does not correspond to the current clause number then it is skipped. if it corresponds then the value of the current term is obtained from the binding of the current variables and unification with the current clause is tried. If unification fails then Clause_list and Clause number are advanced. If it succeeds then it proceeds as follows.

Assume the k+1-th term of the current clause is to be expanded at the state s and alternative clauses for the term are expanded and replaced by backtracking so that this term is passed n times in success. At the same time the first alternative clause is assumed to be passed n₁ times in success.

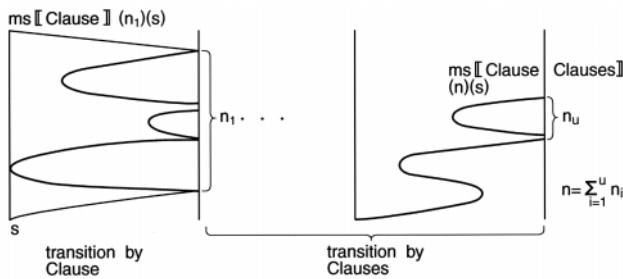


Fig.4 The state transition by the clauses.

Then Clauses, which is the rest of the current Clauses of the term after it is deprived of the first alternative clause, is passed n-n₁ times in success. So the state transition by the rest of the Clauses is written as

$m(s) [[\text{Clauses}]] (n-n_1)(s_1)$

where s₁ is the state before entering into the Clauses. Let m be the total of success passes of the current term of the state before entering into the expansion of the current alternative clause. The number m is obtained from the backtrack_state column of the current k+1-th term description of the partial state where k is found as an index in the process_status column of the state s. So m is written as

$m = \text{the_number_of_success_passes}(s)$.

It is impossible to pass current term with the first alternative clause successfully any more after the n₁ times success passes. The function to obtain n₁ which is written as n(Clause)(s) is defined as

$n(\text{Clause})(s) = F(n+m) - m$

where

$F(t) = \text{if Fail_process}(m(s) [[\text{Clause}]] (t-1)(s))$
 then $F(t-1)$
 else $\text{number_of_success_passes}(m(s) [[\text{Clause}]] (t-1)(s))$.

The state s₁ is obtained from s by

- 1) incrementing next_alternative_clause by 1
- 2) incrementing number_of success_passes by n(Clause)(s)
- 3) setting the number_of_the_most_recent_success_passes to 0

of the partial_state which corresponds to the k+1-th term. This function is written as next_clause(s). So

$s_1 = \text{next_clause}(s)$

Let us define $m(s) [[\text{Clause}]]$

$m(s) [[\text{Clause}]] (n)(s)$

$= \text{if Fail_process_status}(m(s) [[\text{Clause}]] (n-1)(s))$

then $m(s) [[\text{Clause}]] (n-1)(s)$

$\text{if Suc_process_status}(m(s) [[\text{Clause}]] (n-1)(s))$

then $\text{if have_and}(\text{Clause})$ then

$m(s) [[\text{And_part}]] (n)(s)$

else $\text{if } n + \text{the_most_recent_success_passes}(s) > 1$

then $\text{set}(\text{fail})(s)$

else $\text{tran}(\text{Clause})(s)$.

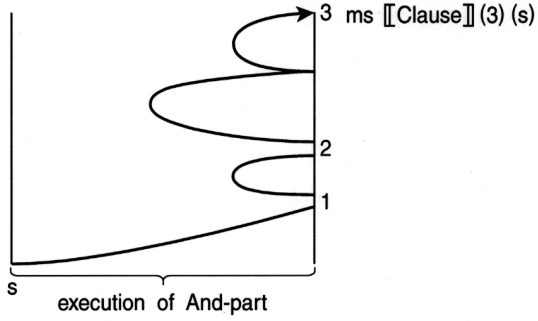


Fig.5 State transition by the clause.

If the maximum number of success passes of the current alternative clause of the state s is less than $n-1$ then $m(s) \llbracket \text{Clauses} \rrbracket (n)(s)$ equals to $m(s) \llbracket \text{Clause} \rrbracket (n-1)(s)$. If it is greater than $n-1$ then the And_part of the Clause is executed.

If the Clause has no And_part then it can be successfully passed at most once. So in case that $n + \text{the_most_recent_success_passes}$ is greater than 1, The process_status of the state s is set to fail and the next_alternative column of the partial state of the current term is incremented by 1. This is the working of set(fail). That is

$$\begin{aligned} \text{set(fail)}(cl, \alpha, \langle a_1, \dots, a_k, \langle cl', \beta, bt \rangle, \dots \rangle, \langle u, k \rangle \rangle \\ = \langle cl, \alpha, \langle a_1, \dots, a_k, \langle cl' + 1, \beta, bt \rangle, \dots \rangle, \\ \langle fail, k \rangle \rangle. \end{aligned}$$

In case of success return of one time, the current variable bindings should be changed according to the current alternative clause. This function tran(Clause)(s) is defined as follows.

$$\begin{aligned} \text{tran(Clause)}(s) = & \text{if Suc(head(Clause))(cur(s))} \\ & \text{then Set_binding}(\llbracket \text{cur_term}(s) \leq \text{cur_bind_pred}(s) \rrbracket (\llbracket \text{head(Clause)} \leq \text{head(Clause)} \rrbracket (\text{cur}(s))) \\ & \quad \cap \text{cur_bind}(s))(s) \\ & \text{else } \perp . \end{aligned}$$

where

$$\begin{aligned} \text{cur_term}(s) &= q_{k+1} \\ \text{cur_bind_pred}(s) &= M_j(x_1, \dots, x_j) \\ \text{cur_bind}(s) &= \beta_1 \\ \text{for } s &= \langle p: -q_1, \dots, q_r, \langle a_1, \dots, a_{k-1}, \\ & \quad \langle cl_1, \beta_1, bt_1 \rangle, \langle cl_2, \beta_2, bt_2 \rangle, \dots \rangle, \\ & \quad \langle u, k \rangle \rangle \end{aligned}$$

and

$$\begin{aligned} \langle u_1, v_1 \rangle \cap \langle u_2, v_2 \rangle &= \langle u_1 \cap u_2, v_1 \cap v_2 \rangle \\ \text{Set_binding}(\beta)(s) &= \langle p: -q_1, \dots, q_r, \alpha, \\ & \quad \langle a_1, \dots, a_{k-1}, \langle cl_1, \beta_1, bt_1 \rangle, \langle cl_2, \beta_2, bt_2 \rangle, \\ & \quad \dots \rangle, \langle u, k+1 \rangle \rangle \end{aligned}$$

The result of unification of head(Clause) and the current term value cur(s) is

$$\llbracket \text{head(Clause)} \leq \text{head(Clause)} \rrbracket (\text{cur}(s)).$$

The function of

$$\llbracket \text{cur_term}(s) \leq \text{cur_bind_pred}(s) \rrbracket$$

is to take up variable binding information from this result. It is not until the intersection of current variable bindings and cur_bind(s) is taken that the next binding is obtained.

$$\begin{aligned} m(s) \llbracket \text{And_part} \rrbracket (n)(s) \\ = & \text{if Suc_index}(m(s) \llbracket \text{terms} \mid \Lambda \rrbracket (n)(s)) \\ & \text{then RTN}(m(s) \llbracket \text{terms} \mid \Lambda \rrbracket (n)(s)) \\ & \text{if Fail_index}(m(s) \llbracket \text{terms} \mid \Lambda \rrbracket (n)(s)) \text{ then} \\ & \quad \text{set(fail)}(s) \\ & \text{if !_Fail_index}(m(s) \llbracket \text{terms} \mid \Lambda \rrbracket (n)(s)) \\ & \quad \text{then set(!fail)}(s) \end{aligned}$$

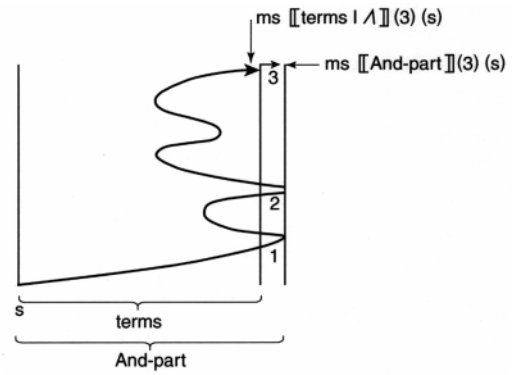


Fig.6 State transition by the And-part.

Assume the process started from the state s and reached the state $s_1 = m(s) \llbracket \text{terms} \mid \Lambda \rrbracket (n)(s)$ after it had passed terms of And_part n times. The most recent binding information of the head of the clause of s_1 is given as

$$Q_1 = \llbracket \text{cur_bind_pred}(s_1) \leq \text{head}(s_1) \rrbracket (\text{cur_bind}(s)).$$

To return back this information to the state s it is necessary to obtain

$$Q_2 = \text{cur_bind}(s) \cap \llbracket \text{cur_term}(s) \leq \text{cur_bind_pred}(s) \rrbracket (Q_1)$$

using cur_term of s . Thus final result s' of applying And_part to s is obtained as

$$s' = \text{Set_binding}(Q_2)(s).$$

s' is written as $\text{RTN}(s_1)(s)$. That is

$$\begin{aligned} \text{RTN}(s_1)(s) &= \text{Set_binding}(Q_2)(s) \\ &= \text{Set_binding}(\text{cur_bind}(s) \cap \llbracket \text{cur_term}(s) \leq \\ & \quad \text{cur_bind_pred}(s) \rrbracket (\llbracket \text{cur_bind_pred}(s_1) \leq \\ & \quad \text{head}(s_1) \rrbracket (\text{cur_bind}(s))))). \end{aligned}$$

Thus former half of $m(s) \llbracket \text{And_part} \rrbracket$ is obtained.

Suc_index(x), Fail_index(x) and !_Fail_index(x) are true when the index of the process_status of x is equal to 1) the length of the list of a partial state, 2) -1 and 3) -2 respectively. The function set(!fail) sets the clause number of the next alternative column of the partial state of the curterm of the state to the number of the last alternative clause +1 and then it performs set(fail).

In case that terms $\neq \Lambda$

$$m(s) \llbracket \text{terms term} \mid \text{terms}' \rrbracket (n)(s)$$

$$= \text{if Suc_process_status}(m(t) \llbracket \text{term} \mid \text{terms}' \rrbracket (r+1)(m(s) \llbracket \text{terms} \mid \text{term terms}' \rrbracket (k)(s)))$$

$$\text{then } m(s) \llbracket \text{term} \mid \text{terms}' \rrbracket (r+1)(m(s) \llbracket \text{terms} \mid \text{term terms}' \rrbracket (k)(s))$$

$$\sqcup \text{if Fail_process_status}(m(t) \llbracket \text{terms} \mid \text{term terms}' \rrbracket (k+1)(s))$$

$$\text{then } m(t) \llbracket \text{terms} \mid \text{term terms}' \rrbracket (k+1)(s)$$

$$\text{else}$$

$$m(t) \llbracket \text{term} \mid \text{terms}' \rrbracket (1)(m(s) \llbracket \text{terms} \mid \text{term terms}' \rrbracket (1)(s))$$

where

$$k = \text{number_of_previous_success_passes}(s)$$

$$r = \text{number_of_the_most_recent_success_passes}(s)$$

$$l = \min_y [k+1 \leq y, \text{Suc_process_status}(m(t) \llbracket \text{term} \mid \text{terms}' \rrbracket (1)(m(s) \llbracket \text{terms} \mid \text{term terms}' \rrbracket (y)(s))]$$

$$\min_y F(y) = \varphi(1)$$

$$\varphi(k) = \text{if } F(k) \text{ then } k$$

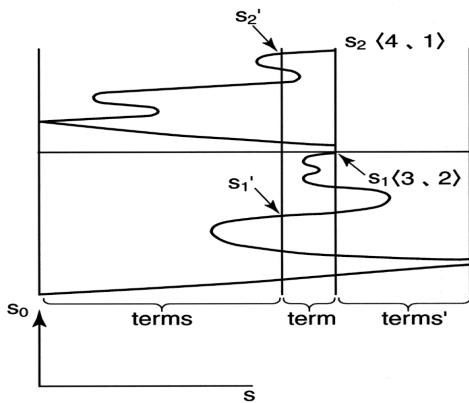
$$\text{else } \varphi(k+1)$$


Fig.7 State transition by the $m(s) \llbracket \text{terms term} \mid \text{terms}' \rrbracket$

Let s_1, s_2 be as

$$s_1 = m(s) \llbracket \text{terms term} \mid \text{terms}' \rrbracket (n-1)(s)$$

$$s_2 = m(s) \llbracket \text{terms term} \mid \text{terms}' \rrbracket (n)(s).$$

To obtain s_2 from s_1 is to be considered.

The state s_1 can be written as

$$s_1 = \langle cl, \alpha, \langle a_1, \dots, a_{k-1}, \langle cl_1, \alpha_1, \langle k, r_1 \rangle \rangle, \langle cl_2, \alpha_2, \langle n-1, r \rangle \rangle, \dots \rangle, \langle u, ix \rangle \rangle.$$

From this description it can be seen that the constructive terms from the first to the ix-th was successfully passed k times and the constructive terms from the first to the ix+1-th was successfully passed n-1 times. Further it is found that the number of the most recent success passes of the consecutive terms from the first to the ix+1-th is r. Thus we write

$$k = \text{number_of_previous_success_passes}(s).$$

The number r is obtained as

$$r = \text{number_of_the_most_recent_success_passes}(s).$$

To obtain s_2

$$1) \text{ call } m(s) \llbracket \text{terms} \mid \text{term terms}' \rrbracket (k)$$

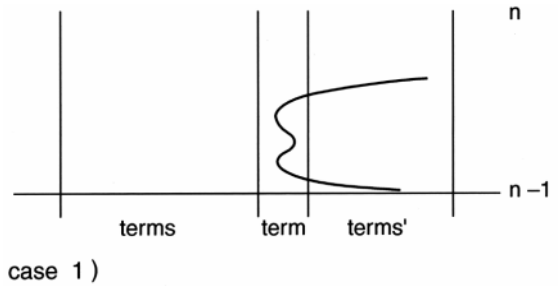
$$\text{and call } m(t) \llbracket \text{term} \mid \text{terms}' \rrbracket (r+1).$$

If $m(s) \llbracket \text{term} \mid \text{terms}' \rrbracket (r+1)$ results in fail then

$$2) \text{ call } m(s) \llbracket \text{terms} \mid \text{term terms}' \rrbracket (k+1)$$

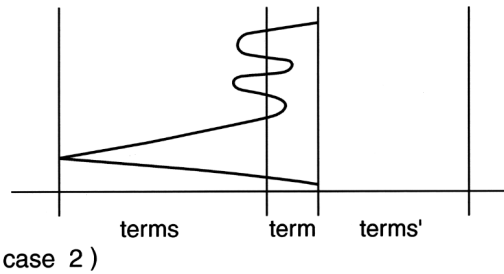
$$\text{and call } m(t) \llbracket \text{term} \mid \text{terms}' \rrbracket (1).$$

If $m(s) \llbracket \text{terms} \mid \text{term terms}' \rrbracket (k+1)$ results in fail then stop else if $m(t) \llbracket \text{term} \mid \text{terms}' \rrbracket (1)$ results in fail increment k to k+1 and go to 2).



case 1)

Fig.8.1 Call $m(s) \llbracket \text{terms term} \mid \text{terms}' \rrbracket (k)$ and call $m(t) \llbracket \text{term} \mid \text{terms}' \rrbracket (r+1)$



case 2)

Fig.8.2 Call $m(s) \llbracket \text{terms term} \mid \text{terms}' \rrbracket (k+1)$ and call $m(t) \llbracket \text{term} \mid \text{terms}' \rrbracket (1)$

In case that terms = Λ

$$m(s) \llbracket \text{terms term} \mid \text{terms}' \rrbracket (n)(s)$$

$$= m(t) \llbracket \text{term} \mid \text{terms}' \rrbracket (n)(\langle \text{cur_clause}(s), \llbracket \text{cur_term}(s) \leq \text{head}(\text{cur_clause}(s)) \rrbracket (\text{cur}(s)) \rangle,$$

$\text{init}(\text{And_part}(\text{cur_clause}(s)), \langle \text{suc}, 0 \rangle)$
 where $\text{And_part}(p: -q_1, \dots, q_r) = q_1, \dots, q_r$

$$\text{init}(q_1, \dots, q_r) = \begin{cases} \langle 1, \text{NIL}, \langle 0, 0 \rangle \rangle \text{init}(q_2, \dots, q_r) & \text{in case } q_1 \text{ is a term} \\ \langle !, \text{NIL} \rangle \text{init}(q_2, \dots, q_r) \text{ else} \end{cases}$$

$$\text{m}(s) \llbracket \text{terms} \mid \text{terms}' \rrbracket (n)(s)$$

$$= \text{if } n=1 \text{ then shift}(\text{m}(s) \llbracket \text{terms} \mid \text{terms}' \rrbracket (n)(s))$$

$$\text{else } \langle \text{cur_clause}(s), \llbracket \text{cur_term}(s) \rangle =$$

$$\text{head}(\text{cur_clause}(s)) \llbracket (\text{cur}(s)),$$

$$\text{init}(\text{And_part}(\text{cur_clause}(s))), \langle \text{fail}, -2 \rangle \rangle$$

where $\text{shift}(\langle \text{cl}, \alpha, \langle a_1, \dots, a_{k-1}, \langle n_1, u^1, v^1 \rangle$
 $\langle n_2, u^2, v^2 \rangle, \dots, \langle \gamma, k \rangle \rangle$
 $= \langle \text{cl}, \alpha, \langle a_1, \dots, a_{k-1}, \langle n_1, u^1, v^1 \rangle, \langle n_2, u^2, v^2 \rangle, \dots, \langle \gamma, k+1 \rangle \rangle$
 $\text{m}(t) \llbracket \text{term} \mid \text{terms} \rrbracket (n)(s)$
 $= \text{if } \text{Suc_process_status}(\text{m}(s) \llbracket \text{Clause_group}(\text{term}) \rrbracket (n)(s)) \text{ then increment_index}(\text{replace_partial_state}(\text{set}(\text{suc}(s))(\text{former_partial_states}(\text{m}(s) \llbracket \text{Clause_group}(\text{term}) \rrbracket (n)(s))(\text{cur_index}(s) + 1) \llbracket \text{latter_partial_states}(\text{tran}(\llbracket \text{terms} \rrbracket (\text{m}(t) \llbracket \text{term} \mid \text{terms} \rrbracket (n-1)(s))) (\text{index}(s) + 2)) \text{ else decrement_index}(\text{set}(\text{fail}(s))$

where
 $\text{increment_index}(s) = \langle \text{cl}, \alpha, \langle a_1, \dots, a_k, \langle n, u, \beta \rangle, \dots, \langle c, k+1 \rangle \rangle$
 $\text{former_partial_states}(s)(l) = a_1, \dots, a_l$
 $\text{latter_partial_states}(s)(l) = a_{l+1}, \dots, a_{\text{length}(\text{And_part}(\text{cl}))}$
 $\text{decrement_index}(s) = \langle \text{cl}, \alpha, \langle a_1, \dots, a_k, \langle n, u, \beta \rangle, \dots, \langle c, k-1 \rangle \rangle$
 for $s = \langle \text{cl}, \alpha, \langle a_1, \dots, a_k, \langle n, u, \beta \rangle, \dots, \langle c, k \rangle \rangle$
 and
 $\text{replace_partial_states}(\langle \text{cl}, \alpha, \gamma, \beta \rangle) (\delta)$
 $= \langle \text{cl}, \alpha, \delta, \beta \rangle$

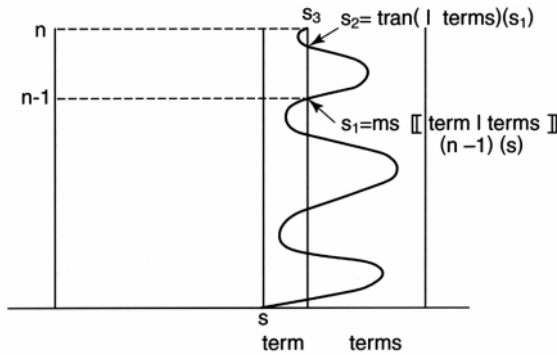


Fig.9 State transitions from S to S₁, S₁ to S₂ and S₂ to S₃.

Assume that the process transited from s to s_1 by the application of $\text{m}(t) \llbracket \text{term} \mid \text{terms} \rrbracket$ and that it further went and turned back to s_2 by the application of $\text{tran}(\llbracket \text{terms} \rrbracket)$ and then reached s_3 by the application of $\text{m}(s) \llbracket \text{Clause_group}(\text{term}) \rrbracket$

Thus
 $s_1 = \text{m}(t) \llbracket \text{term} \mid \text{terms} \rrbracket (n-1)(s)$
 $s_2 = \text{tran}(\llbracket \text{terms} \rrbracket)(s_1)$

The state s_3 is the same to the one which is obtained from $\text{m}(s) \llbracket \text{Clause_group}(\text{term}) \rrbracket (n)(s)$ by replacing the latter part of its partial status list from the current partial status with that of s_2 . Here $\text{Clause_group}(\text{term})$ denotes the Clause_group corresponds to the term. Finally let's define $\text{tran}(\llbracket \text{terms} \rrbracket)$

$$\text{tran}(\llbracket \text{terms} \rrbracket)(s) = \text{m}(t) \llbracket \text{term} \mid \text{terms} \rrbracket (\min_y \text{Fail_process_status}(\text{m}(t) \llbracket \text{term} \mid \text{terms} \rrbracket (y)(s)))(s)$$

§3 Prolog program on the space E_0

In section 2. it is shown that Prolog program is defined and continuous on the space E . However the main concern of Prolog program is its behavior on the data domain E_0^* . If Prolog program is not defined almost every where on E_0^* then the functional interpretation of section 2 is meaningless. So in this section the definability of Prolog program on E_0^* shall be discussed.

The functions which directly operate on the element of E in the definition of Prolog program as a function on E of section 2 are

$$\llbracket p(x,y) \leq q(x,z) \rrbracket \cup \text{ and } \cap$$

It is unclear that the latter two functions are totally defined on $E_0^* \times E_0^*$. The functions \cup and \cap are equivalent to \cup and \cap in their positive part respectively. The function \cup is totally defined and continuous on $E_0^* \times E_0^*$. While the function \cap is only lower semi-continuous on $E_0^* \times E_0^*$. If \cap is defined totally on $E_0^* \times E_0^*$ then \cap must be equal to \cap on $E_0^* \times E_0^*$. So \cap must be continuous on $E_0^* \times E_0^*$. This is a contradiction. This means \cap is not defined everywhere on $E_0^* \times E_0^*$. We need not adhere to \cap and \cup . If there are continuous functions \cap' and \cup' such that $\cap' = \cap, \cup' = \cup$ almost everywhere on $E_0^* \times E_0^*$ then we can replace \cup and \cap by \cup' and \cap' respectively.

In this section the existence of such functions \cup and \cap shall be shown.

[Definition 8]

Let γ, α be a finite subset of FT(N) and β can be written as $\beta = \bar{b}(\beta') \cap FT(N)$ for some finite subset β' of FT. Then step function $\chi_{\alpha, \beta}^{\gamma}(x)$ is defined as follows.

$$\chi_{\alpha, \beta}^{\gamma}(x) = \begin{cases} \gamma & \text{in case } x \supset \alpha \text{ and } x \cap \beta = \emptyset \\ \emptyset & \text{else} \end{cases}$$

[Proposition 4]

$\chi_{\alpha, \beta}^{\gamma}$ is continuous on E_0^* .

(Proof)

$U(\alpha, \beta)$ is a clopen set. So it is clear.

Q.E.D.

[Proposition 5]

$x \cup y$ is continuous on $E_0^* \times E_0^*$.

(Proof)

Let α be a finite subset of FT(N) and β' be a finite subset of FT such that

$$x \cup y \in U(\alpha, \beta)$$

for $\beta = \bar{b}(\beta')$

We can select α_1 and α_2 such that

$$\alpha = \alpha_1 \cup \alpha_2, x \supset \alpha_1 \text{ and } y \supset \alpha_2.$$

Thus

$$x \in U(\alpha_1, \beta) \text{ and } y \in U(\alpha_2, \beta).$$

Further

$$u \cup v \in U(\alpha, \beta)$$

for all $u \in U(\alpha_1, \beta)$ and $v \in U(\alpha_2, \beta)$.

Q.E.D.

[Definition 9]

A function f on E_0^* is said to be lower semi-continuous on E_0^* if and only if for any $x \in E_0^*$ and for any finite subset α of FT(N) there exists U , a neighborhood of x such that for any $y \in U$

$$f(x) \supset \alpha \Rightarrow f(y) \supset \alpha$$

holds.

[Proposition 6]

Let $\{f_n\}$ be a set of continuous functions on E_0^* . Then the function $\bigcup_{n \geq 1} f_n$ is lower semi-continuous on E_0^* .

(Proof)

Let α be a finite subset of FT(N) and $U(\alpha, \emptyset)$ satisfies

$$\bigcup_{n \geq 1} f_n(x) \in U(\alpha, \emptyset).$$

It is easy to see that

$$\bigcup_{n \leq N_0} f_n(x) \in U(\alpha, \emptyset)$$

for some natural number N_0 . From proposition 5 $\bigcup_{n \leq N_0} f_n$ is continuous. So there is a neighborhood $U(\gamma, \delta)$ of x such that

$$\bigcup_{n \leq N_0} f_n(y) \in U(\alpha, \emptyset)$$

for any $y \in U(\gamma, \delta)$.

This shows

$$\left(\bigcup_{n \geq 1} f_n \right) (U(\gamma, \delta)) \subset U(\alpha, \emptyset).$$

Q.E.D.

[Proposition 7]

If f is lower semi-continuous then it can be written as

$$f = \bigcup \chi_{\alpha, \beta}^{\gamma} \\ \chi_{\alpha, \beta}^{\gamma} \subset f$$

(Proof)

Let x be an arbitrary element of E_0^* . Since f is lower semi-continuous there exists a finite subset γ of FT(N) and a finite subset δ' of FT for any finite subset α_0 of $f(x)$ such that

$$x \in U(\gamma, \delta) \quad f(y) \supset \alpha_0 \quad \text{for any } y \in U(\gamma, \delta)$$

for $\delta = \bar{b}(\delta') \cap FT(N)$.

That is $f \supset \chi_{\gamma, \delta}^{\alpha}$.

This implies

$$f(x) \supset \bigcup \chi_{\gamma, \delta}^{\alpha} (x) \supset \alpha_0. \\ \chi_{\gamma, \delta}^{\alpha} \subset f$$

There is a family $\{\alpha_i\}$ of finite subsets of FT(N) such that

$$f(x) = \bigcup_{i \geq 1} \alpha_i.$$

So

$$f(x) \supset \bigcup \chi_{\gamma, \delta}^{\alpha} (x) \supset \bigcup_{i \geq 1} \alpha_i = f(x) \\ \chi_{\gamma, \delta}^{\alpha} \subset f$$

Q.E.D.

[Definition 10]

Let α, γ be finite subsets of FT(N), β' be a finite subset of FT and $\beta = \bar{b}(\beta') \cap FT(N)$.

Define $\eta_{\alpha, \beta}^{\gamma}$ as

$$\eta_{\alpha, \beta}^{\gamma}(x) = \begin{cases} \gamma^c & \text{in case } x \in U(\alpha, \beta) \\ FT(N) & \text{else} \end{cases}$$

[Proposition 8]

If f is a lower semi-continuous function on E_0^* then the closure of the set

$$\{ x \mid f(x) \neq \bigcap \eta_{\alpha,\beta}^\gamma(x) \}$$

does not include any non-empty open set.

(Proof)

Let U be

$$U = \{ x \mid f(x) \neq \bigcap \eta_{\alpha,\beta}^\gamma(x) \}$$

For any $x \in U$ we can select $c(x)$ of $FT(N)$ such that

$$f(x) \not\supseteq c(x) \text{ and } \bigcap \eta_{\alpha,\beta}^\gamma(x) \supseteq c(x).$$

Assume the closure of U includes some non-empty open set. Then there exists a finite subset α_0 of $FT(N)$ and β_0' of FT such that

$$U(\alpha_0, \bar{b}(\beta_0') \cap FT(N)) \subset \bar{U}.$$

The set $\{ \overline{c^{-1}c(x)} \mid x \in U \}$ is countable. So it can be written as

$$\{ \overline{c^{-1}c(x)} \mid x \in U \} = \{ V_n \mid n \geq 0 \}.$$

If there exists n_1 such that

$$V_{n_1} \supset U(\varepsilon, \delta) \text{ and } \delta = \bar{b}(\delta') \cap FT(N)$$

for some finite subset ε of $FT(N)$ and δ' of FT , then there exists $x_0 \in U$ such that $c^{-1}c(x_0)$ is dense in $U(\varepsilon, \delta)$. Because x_0 can be selected so that

$$\overline{c^{-1}c(x_0)} = V_{n_1}$$

is satisfied. From the lower semi-continuity of f ,

$$c(x_0) \not\supseteq f(x) \text{ for any } x \in U(\varepsilon, \delta)$$

holds.

This means

$$\eta_{\varepsilon,\delta}^{\{c(x_0)\}} \supset f.$$

This implies

$$\eta_{\alpha,\beta}^\gamma \supset f \quad \eta_{\alpha,\beta}^\gamma(x) \not\supseteq c(x_0)$$

It contradicts. This shows V_n does not include any non-empty open set for any $n \geq 0$. For $n=0$ this means

$$U(\alpha_0, \beta_0) \setminus V_0 \neq \emptyset.$$

$U(\alpha_0, \beta_0) \setminus V_0$ is open because V_0 is closed. So there must exist an open set $U(\alpha_1, \beta_1)$ such that

$$U(\alpha_1, \beta_1) \subset U \setminus V_0.$$

Here α_1 is a finite subset of $FT(N)$ and β_1 can be written as $\beta_1 = \bar{b}(\beta_1') \cap FT(N)$ for some finite subset β_1' of FT . Applying this procedure consecutively, the sequence of non empty open set $U(\alpha_n, \beta_n)$ such that

$$U(\alpha_n, \beta_n) \subset U(\alpha_{n-1}, \beta_{n-1}) \setminus V_{n-1}$$

can be obtained. If we assume

$$\bigcap U(\alpha_n, \beta_n) = \emptyset$$

then from its definition

$$\bigcup_{n \geq 1} \alpha_n \cap \bigcup_{i \geq 1} \beta_n \neq \emptyset.$$

So there exists an element a of $FT(N)$ such that

$$a \in \alpha_{n_0} \cap \beta_{n_1}$$

for some n_0, n_1 .

From the monotone increasing property of α_n and β_n there exists N_0 such that

$$a \in \alpha_{n_0} \cap \beta_{n_0}$$

This means

$$U(\alpha_{n_0}, \beta_{n_0}) = \emptyset.$$

This contradicts.

That is

$$\bigcap_{n \geq 1} U(\alpha_n, \beta_n) \neq \emptyset.$$

While

$$\begin{aligned} \bigcap_{n \geq 1} U(\alpha_n, \beta_n) &\subset \bigcap_{n \geq 1} (U(\alpha_n, \beta_n) \setminus V_n) \\ &\subset U \setminus \bigcup_{n \geq 1} V_n = \emptyset. \end{aligned}$$

This contradicts.

Q.E.D.

[Definition 11]

Let γ and α be a finite subset of $FT(N)$ and β' be a finite subset of FT . Then the function

$\tau_{\langle \alpha, \bar{b}(\beta') \rangle}^\gamma$ on E is defined as

$$\tau_{\langle \alpha, \bar{b}(\beta') \rangle}^\gamma (\langle u, v \rangle) = \begin{cases} \gamma & \text{in case } u \supset \alpha \text{ and} \\ & v \supset \bar{b}(\beta') \\ \emptyset & \text{else} \end{cases}$$

Let f be a lower-semi-continuous function on E_0^* . Then the function \hat{f} on E is defined as

$$\hat{f} = \bigcup_{f \supset \chi_{\alpha, \bar{b}(\beta') \cap FT(N)}^\gamma} \cup \tau_{\langle \alpha, \bar{b}(\beta') \rangle}^{\langle \gamma, \phi \rangle} \cup \bigcup_{f \subset \chi_{\alpha, \bar{b}(\beta') \cap FT(N)}^\gamma} \tau_{\langle \alpha, \bar{b}(\beta') \rangle}^{\langle \phi, \gamma \rangle}$$

[Theorem 1]

Let f be a lower semi continuous function on E_0^* . Then the closure of the set $\{ x \mid x \in E_0 \text{ and } \hat{f}(x) \neq I^{-1} fI(x) \}$ does not include any non-empty open set of E_0 .

(Proof)

Let U be a subset of E_0 defined as

$$U = \{ x \mid f(x) = \bigcap \eta_{\alpha,\beta}^\gamma(x) \text{ and for some } \eta_{\alpha,\beta}^\gamma \supset f$$

$$y, z \langle f(x), z \rangle \in E_0, \langle x, y \rangle \in E_0 \text{ and } \langle f(x), z \rangle \neq \hat{f}(\langle x, y \rangle) \}.$$

It is sufficient to show that the closure of U does not include any non-empty open set. For any $x \in U$ there exists $y, z \in P(FT)$ such that

$$\langle f(x), z \rangle \neq \hat{f}(\langle x, y \rangle) \text{ and } \langle f(x), z \rangle, \langle x, y \rangle \in E_0.$$

Because of the fact that

$$\hat{f}(\langle x, y \rangle) = \langle u, v \rangle \neq \langle f(x), z \rangle$$

and

$$u = \cup \chi_{\alpha, \beta}^{\gamma}(x) = f(x), v \subset z$$

$$\chi_{\alpha, \beta}^{\gamma} \subset f$$

there must exist $c \in FT$ such that

$$\bar{b}(c) \subset z \text{ and } \bar{b}(c) \not\subset v$$

$\bar{b}(c)$ can be written as $c(x)$.

The set $\{c(x) \mid x \in U\}$ is countable. So it can be written as $\{c_n \mid n \geq 1\}$.

Let U_n and V_n be

$$U_n = \{x \mid c(x) = c_n \text{ and } x \in U\}$$

and $V_n = \bar{U}_n$.

The similar argument of Proposition 8 shows this theorem.

Q.E.D.

The result of theorem 1 can be easily extended to the lower semi-continuous functions of several variables. While \cup and \cap are lower semi-continuous functions on $E_0^* \times E_0^*$. So \cup and \cap coincides with some continuous functions \cup' and \cap' on $E \times E$ respectively almost everywhere on $E_0^* \times E_0^*$. From Scott⁴⁾, Lambda definability is equivalent to the recursive enumerability of the graph of a function. So constructing \cup' and \cap' according to the definition 11 it is easily seen that they can be made Lambda definable.

§4 Conclusions

There could be many arguments on the semantics of Prolog. But I believe the two targets, the first one to localize the state description and the second one to preserve set function interpretation, are achieved in this paper. Thus this approach can be positioned between the Kowalsky, Emden's²⁾ and the Komorowski⁸⁾, Johnes-Mycroft's⁹⁾. There remains a problem of how to interpret assert statements. An assert statement changes the connection of clauses to the clause_group. So the problem is how to represent the connection. It is easy to transform the state transition model in this paper to the continuation model. It appears meaningful in the case of describing files. But even in the case the information on an input file should be kept in the state description. This can be avoided only by treating input as a stream as Carlson¹⁰⁾.

The important advantage of our approach are that it

can be easily extended so as to give semantics to so called set-of construct and that it provides us with easier means of inductive method in proving properties of Prolog programs. The rescent approaches of denotational description do not require continuity. The reason is that the only important point is to give semantics for recursion and not to give semantics for higher order functions, because the languages such as Pascal does not require higher order functions. In Prolog case it also holds. But I want precise semantics and to keep reality as much as possible.

To define graph function on E , such that the recursive combinator coincides with the least fixed point operator, is possible. Using such graph model \cup' and \cap' can be directly shown to be Lambda definable. But it is not so important. The graph model is interesting only in the case that there is some property which cannot be shown only by the syntax of λ -Calculus as in Barhrendregt⁷⁾.

References

- 1) K.R.Apt and M.H.van Emden "Contributions to the theory of logic programing", JACM 29 (1982) 841-862.
- 2) M.H.van Emden and R.A.Kowalski "The semantics of predicate logic as a programing language", JACM 23 (1976) 733-742.
- 3) J-L.Lassez and M.J.Maher "Optimal fixed points of logical programs" 3rd Conf. on Foundations of Software technology and Theoretical computer science, Dec. 1983.
- 4) D.Scott "Calculus and recursion theory" Proc. 3rd Scandinavian Logic Symp. North Holland 154-193.
- 5) D.Scott "Data type as lattices" SIAM J. Computing 5 3 September 1976.
- 6) G.D.Plotkin "T^ω as a universal domai" J.Compt.and System Science 17 209-236.
- 7) Bahrendregt "Equality of lambda terms in the model T^ω" To H.B.Curry:Essays on combinatory logic, lambda calculus and formalism, Academic press (1980) 303-337.
- 8) Komorowski, H.J. "A specification of an Abstract Prolog Machine and Its Application to Partial Evaluation" Ph.D.Thesis, Linkoping

University Linkoping (1981).

- 9) Johnes N.D. and Mycroft A. "Stepwise development of operational and denotational semantics for Prolog" Proceedings of the 2nd logic Programing Conference.
- 10) Carlsson M. "On Implementing Prolog in Functional Programing" New Generation Computing 2 (1984) 347-359.
- 11) Stoy J.E. "Denotational Semantics : The Scott Strarchy Approach to Programing Language Theory" M.I.T.Press Cambridge Mass.

(Accepted March 12, 1998)