PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. 



# A NOTE ON THE SET ABSTRACTION IN LOGIC PROGRAMMING LANGUAGE

Takashi Yokomori

International Instutite for Advanced Study of
Social Information Science, Fujitsu Limited
Numazu, Shizuoka 410-03 JAPAN

**ABSTRACT**

The concept of set abstraction is introduced as a simple analogy of that of lambda abstraction in the theory of lambda calculus. The set abstraction is concerned with two extensions concerning PROLOG language features : "set expression" and "predicate variable". It has been argued in literature that the set expression extension to PROLOG does really contribute to the power of the language, while the extension of predicate variables does not add anything to PROLOG.

Combining these two concepts of extensions to PROLOG, we define "set abstraction" as the set expression in which predicate variables are allowed as data objects. In other words, the set abstraction gets involved in the higher order predicate logic. It is demonstrated that with the help of predicate variables set abstractions can nicely handle the world of the second order predicate logic. Further, the implementaion programs written in PROLOG and Concurrent Prolog are presented.

## 1. Introduction

Since a class of formulae in the first order predicate logic called Horn clauses has been shown to be quite useful by Kowalski in that it can provide with an interesting computation model, a programming language PROLOG has been receiving much attention and has been intensively studied. A Horn clause program is often called a "Pure Prolog" program in which no illogical construct is allowed, while a practical language PROLOG may contain a control primitive like the "cut" operator and other primitives to extend its language capability. Among those, the set expression extension to PROLOG has been often argued and implemented in several languages. For example,there are predicates "setof" in DEC-10 Prolog(Bowen 1981),"set" in PARLOG(Clark and Gregory 1983), "collect" in LM-Prolog(Kahn 1984), and "enumerate" in KL1(Furukawa et al. 1984). The IC-Prolog(Clark et al. 1982) also allows the set expression in a query. The introduction of set expressions enables one to describe the set of all solutions to some goal in a program. As Warren discussed in his paper(Warren 1981), the extension of set expressions to PROLOG really contributes to the power of the language. In the paper above, besides set expressions he also focused on two possible "higher order" extensions to PROLOG : "predicate variable" and "lambda abstraction", and stressed that these extensions do not add any extra power to PROLOG.

This paper is motivated by Warren's paper above. The purpose of this note is to discuss a possible extension to PROLOG called "set abstraction" and to demonstrate the usefulness of the extension. The set abstraction discussed here can be regarded as an extension of the set expression in which predicate variables are allowed as data objects. It may be also possible to take the set abstraction as a simple analogy of the lambda abstraction. Thus, in this paper we take the position to distinguish the set abstraction from the set expression.

The concept of set abstraction is introduced and the predicate "enumerate" is proposed in Section 2. The predicate enumerate considered here is an extension of the one introduced and discussed in reference to KL1(Furukawa et al. 1984). Section 3 presents the implementation issue of the predicate enumerate. Finally, discussion and concluding remarks are given in Section 4.

The reader is assumed to be familiar with the rudiments of PROLOG.

## 2. Set Abstraction

As mentioned in the previous section, one can introduce the concept of the set abstraction in a natural way. The set abstraction discussed in this paper is a simple analogy of lambda abstraction in the theory of lambda calculus. One may obtain a function from a term by means of lambda abstraction, while with the concept of set abstraction one can associate a relation implied by the term.

Let P be a term contaning free occurrences of a variable x, where the prime functor of P is a predicate symbol. Then, analogously to lambda abstraction, one can define the concept

of set abstraction in the following manner : Using a pair of braces {} instead of a greek letter lambda and paying attention to x free in P, an expression

{}x.P

is called set abstraction, and its intended interpretation is the set of all terms (instances of x) satisfying the relation implied by P. As a notation, we write

{ x | P }

for {}x.P.

Suppose that , for example, a term

have_property(x,P)

meaning that x has a property P is given. By paying attention to x, one may have

{}x.have_property(x,P).

Or, if P is taken for the object of abstraction,

{}P.have_property(x,P)

is obtained.
The former, {x| have_property(x,P)} in the equivalent form, is nothing but the set of all x's having the property P. On the other hand, the latter has more meaningful flavor. When dealing with predicates as data objects like in

{P| have_property(x,P) },

one immediately gets involved in the second order predicate logic, and that is what we are going to put great emphasis on through the discussion in this paper.

In the sequel we argue that the set abstraction extension to PROLOG does really add something new to the language. In the paper by Warren (Warren 1981) he discussed the benefits of introducing the concept of "predicate variables( or predicates as data objects)", "set expression", and "lambda expression" and concluded that presicate variables and lambda expression can be merely regarded as "syntactic sugar" and that they do not increase the real power of PROLOG, while set expressions do indeed fill a real gap in the language.

We shall demonstrate the usefulness of set abstraction extension to PROLOG. Set abstractions considered here is concerned with two extensions to the language: predicate variables and set expression. As previously defined, the set abstraction here can be taken as the set expression in which the treatment of predicate variables is taken into consideration.

Suppose the following knowledge base is given:

(1)
```
child(jim,mary).
child(tom,mary).
child(mary,nancy).
child(barbara,john).
child(john,nancy).
likes(tom,barbara).
likes(mary,jim).
likes(jim,nancy).
likes(tom,mary).
poorer(tom,mary).
poorer(barbara,mary).
poorer(mary,jim).
```

where child(X,Y), likes(X,Y), and poorer(X,Y) mean that X is a child of Y, X likes Y, and X is poorer than Y, respectively.

(2)
```
parent(X,Y) <-- child(Y,X).
ancestor(X,Y) <-- parent(X,Y).
ancestor(X,Y)<--parent(X,Z),ancestor(Z,Y).
brother(X,Y) <-- parent(Z,X),parent(Z,Y),
                 not(identity(X,Y)).
cousin(X,Y) <-- parent(Z,X), parent(W,Y),
                brother(Z,W).
richer(X,Y) <-- poorer(Y,X).
richer(X,Y) <-- poorer(Z,X), richer(Z,Y).
```

where P(X,Y) means that X is a P of Y, for each P in {parent, ancestor, brother, cousin}, identity(X,Y) denotes that X is identical to Y, richer(X,Y) denotes that X is richer than Y.

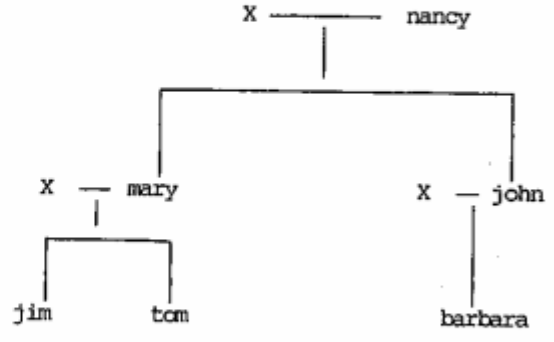


**Figure.** A Family Tree

The Figure above illustrates a family tree in the knowledge base given.

We are now in a position to introduce a predicate for set abstraction. The predicate, we name it "enumerate", has the following syntax and semantics :

Syntax. enumerate(G,L)
where G is a set in either extensional or intensional expression, L is a variable. In the intensional expression, G is of the form {X|conditions}, X is a term with variables, conditions are given as a sequence of goals in Pure Prolog.

Semantics. enumerate(G,L) succeeds if and only if G is nonempty. L serves as a stream variable to bind elements of G if G is infinite. Otherwise, L is a list variable to obtain all elements of G.

In this paper we are mainly concerned with the case when G is finite, while the implementation for the predicate is given for both cases later.

Suppose that one wants to get all pairs (X,Y) such that X is a parent of Y. The procedure to be invoked is :

```
?- enumerate({(X,Y)| parent(X,Y)}, L).
```

The answer to this question is obtained as a list:

```
L = [(john,barbara),(mary,jim),(mary,tom),
     (nancy,john), (nancy,mary)].
```

In the similar way, the respose to the question

```
?- enumerate({ X | cousin(X,barbara) }, L)
```

will be

```
L = [ jim, tom ].
```

That is,it is seen that "jim" and "tom" are cousins of "barbara".

Another type of the usage of enumerate(G,L) demonstrates the usefulness of the predicate, which distinguishes itself from other set predicates proposed and implemented in literature. Suppose that one would like to know the relation between people. For example, if one wishes to list up all relations holding between, say, "tom" and "mary", then the query will be

```
?- enumerate( { P | P(tom,mary) }, L).
```

One will get the response

```
L = [ child, likes, poorer ].
```

Furthermore, for the query

```
?- enumerate({ P | P(mary,tom) }, L)
```

the response

```
L = [ ancestor, parent ,richer ]
```

is obtained. It should be remarked that there is originally no fact of the form P(mary,tom) in the knowledge base at issue. Furthermore, the query

```
?- enumerate({ P | P(jim,barbara) }, L)
```

brings the response

```
L  = [ cousin, richer ].
```

Note that there is no fact of relation between "jim" and "barbara". In other words, attributes "richer" and "cousin" concerning "jim" and "barbara" are the derived results from the knowledge base.
Thus, the predicate "enumerate" achieves the higher order inference function.

Besides these functions, the predicate "enumerate" also has a capability to handle an infinite set. For example, if one make a query
?- enumerate({ X | prime_number(X)}, L),
then one can obtain an infinite stream of prime numbers :

```
2, 3, 5, 7, 11, 13, . . .
```

This kind of approach to handle an infinite set has already taken in recent papers(Clark and Gregory 1983, Furukawa et al. 1984).

Application Examples.

It is easy to extend "enumerate" so that the conditions part may allow arbitrary PROLOG program goals. In fact, the DEC-10 implementation program for the predicate enumerate is given as such later (see Section 3).

(1)Cross reference
Consider the following program :

```
cross_ref(Program,Attribute,L):-
enumerate({P|clause_body(Program,Attribute,P)},
L).

where clause_body(Prog,A,Pred):-
          Q=..[A,_], Y=..[Prog,(P1<--X)],Y,
          P1=..[Pred,_], member_term(Q,X).
```

The predicate cross_ref(P,A,L) takes a program name P and a predicate name A as inputs and outputs a list L of attribute names referring A

in P. For example, provided that the knowledge base considered above is named "ax", a query

```
?-cross_ref(ax,parent,L)
```

would produce

```
L = [ ancestor,brother,cousin ].
```

(2) Higher order sets

The second order set {L|L is the set of relations valid for some common instance in the knowledge base above, and the cardinality of L is 2} is obtained by making a query :

```
?-enumerate({L|X^(enumerate({P|P(X)},L),
                    length(L,2))},S).
```

```
S=[[ancestor,parent],[brother,richer],[child,
   richer],[cousin,likes],[cousin,richer]].
```

### 3. Implementation Examples

Two implementation programs for the predicate "enumerate" are given in this section. One is written in DEC-10 Prolog(Bowen 1981), while the other in Concurrent Prolog(Shapiro 1983). Both programs can run on DEC 2060 system.

The DEC-10 Prolog, like many other languages of PROLOG family, has no facility to support the function of dealing with predicate variables. Another difficulty in implementing the predicate under the circumstance mentioned above is that set predicates in conventional PROLOG is only concerned with a finite set. Further, in case of Concurrent Prolog implementation one cannot expect the backtracking function in order to collect all solutions to a goal.

Through the task of implementing the predicate enumerate it turned out that
(i) in DEC-10 Prolog implementation, the predicate"setof" is essentially required, and the predicate "demo" plays an important role, while
(ii) it is crucial for Concurrent Prolog implementation of "enumerate" to achieve the function of the predicate "eager_enumerate" under the circumstance of no backtracking mechanism. (This will be discussed in more detail later.)

These predicates ("demo" and "eager_enumerate") have been already discussed in literature(Kunifuji et al. 1983, Hirakawa and Chikayama 1984) in reference to the work on the Fifth Generation Computer System, and the attempt in this paper proves the usefulness of those concepts.

Notes.

(1) The predicate "enumerate" written in DEC-10 Prolog can handle only the case where the target set is finite. The Concurrent Prolog version , on the other hand, generates a stream of all elements of the set involed. However, the DEC-10 implementation enables one to describe not only Pure Prolog but any other PROLOG programs for set conditions.
(2) There are,in fact, several PROLOG languages in which predicate varialbes are allowed provided that they must have been instanciated at the execution stage. The DEC-10 Prolog, however, does not support even this partial facility. In the implementation programs presented here , an infix operator "holds_for" is used for the purpose of overcoming this weakness and for the uniform treatment of predicates in a program.
(3) The predicate "enumerate" implemeted here is slightly different from the one defined in the previous section in that the implemented specification allows only intensional expression for sets. It is, however, seen that one can easily modify the program so that the full specification may be satisfied.

The top level procedure for "enumerate" is as follows :

```
( in DEC-10 Prolog )

     enumerate({X | Conditions }, L) :-
               transform(Conditions,Goals),
               setof(X,Goals,L).

where
transform(E^Conditions,E^Goals) :-
               transform1(Conditions,Goals).
transform(Conditions,Goals) :-
               transform1(Conditions,Goals).

transform1(((P holds_for X),C),(eval(P,X),G)):-
               var(P),!,transform1(C,G).
transform1(((P holds_forX),C),(demo(ax,Q),G)):-
               atom(P),!,Q=..[P,X],
               transform1(C,G).
transform1((C1,C2),(C1,G)) :-
               transform1(C2,G).
transform1((P holds_for X),eval(P,X)) :-
               var(P).
transform1((P holds_for X),demo(ax,Q)) :-
               atom(P),Q=..[P,X].
transform1(C,C).
```

The main role of the predicate "transform" is to transform the sequence of conditions given as an input into the sequence of goals possibly containing "eval" and "demo" predicates.

The predicate "demo" is an extended version of the one originally proposed by Bowen and Kowalski(Bowen and Kowalski 1982). It has been intensively investigated by Kunifuji et al. (Kunifuji et al. 1983). The predicate demo(ax,P) succeeds if a goal P succeeds in a program named "ax".

The procedure eval(P,X) defined by

```
eval(P,X):- ax(Y),Y=..[(:-),Z,_],
            Z=..[P,X],demo(ax,Z)
```

commits its evaluation to the predicate demo.

( in Concurrent Prolog )

```
enumerate({X|Conditions},L) :-
     prolog(cp_transform(Conditions,Goals))|
     eager_enumerate({X|Goals),L).
```

The role of the predicate cp_transform is essentially the same as that of "transform". (See below for details.) The predicate eval here is slightly different from the one in DEC-10 Prolog implementation :

```
eval(P,X) :-
     P holds_for X, Z=..[P,X], demo(ax,Z).
```

The predicate "eager_enumerate" plays a central role in the Concurrent Prolog implementation. The predicate eager_enumerate(G,L) takes a set G as an input and generates a (possibly infinite) stream of elements of G in an "eager" way. It has been implemented by Hirakawa and Chikayama(Hirakawa and Chikayama 1984) applying the AND-parallel mechanism of Concurrent Prolog to the OR-parallel execution in Pure Prolog. There is, on the other hand, another way of achieving the eager_enumerate function proposed by Kahn(Kahn 1983) in which the OR-parallel mechanism of Concurrent Prolog executes OR-clauses of Pure Prolog in parallel. The Kahn's implementation is used here because of its simplicity as well as efficiency to certain types of programs.

It should be noted that in the Concurrent Prolog implementation example given here , the syntax of the predicate "enumerate" is different from the one proposed previously in that it has only one argument, which comes from the syntax of Kahn's "eager_enumerate". Further, note that besides the original knowledge base named "ax" , the Concurrent Prolog implementation requires another copy of the knowledge base in which a represention " P holds_for X " is used for " P(X) ".

[Prolog Implementation]

```
:-op(200,xfy,'holds_for').
:-op(1200,xfx,'<--').

enumerate({X|Conditions},L):-
        transform(Conditions,Goals),
        setof(X,Goals,L).
enumerate({X|Conditions},L):-
        transform(Conditions,Goals),
        setof(X,Goals,L).

%-------------------------------------------------------

transform(E^Conditions,E^Goals) :-
        transform1(Conditions,Goals).
transform(Conditions,Goals):-
        transform1(Conditions,Goals).

transform1((P holds_for X),eval(P,X)):-
        var(P).
transform1((P holds_for X),demo(ax,Q)):-
        atom(P),
        Q=..[P,X].
transform1(((P holds_for X),C),(eval(P,X),G)) :-
        var(P),!,
        transform1(C,G).
transform1(((P holds_for X),C),(demo(ax,Q),G)):-
        atom(P),!,
        Q=..[P,X],
        transform1(C,G).
transform1((C1,C2),(C1,G)):-
        transform1(C2,G).
transform1(C,C).

eval(P,X):-
        ax(Y),
        Y=..['<--',Z,_],
        Z=..[P,X],
        demo(ax,Z).

%-------------------------------------------------------

demo(World,true).
demo(World,not(P)):-
        metanot(demo(World,P)).
demo(World,(P;Q)):-
        (demo(World,P);demo(World,Q)).
demo(World,(P,Q)):-
        demo(World,P),
        demo(World,Q).
demo(World,P):-
        systemp(P),!,P.
demo(World,P) :-
        metacall(World,(P<--Q),X),
        demo(World,Q).

metanot(P):-
        P,!,fail.
metanot(_).

metacall(W,P,Wp):-
        Wp=..[W,P],Wp.

systemp(nonvar(_)).
systemp(var(_)).
systemp((X=..[Y,Z])).
systemp(nl).
systemp((write(_))).
systemp((k_write(_))).
systemp((X==Y)).
```

[Concurrent Prolog Implementation]

```
:-op(200,xfy,'holds_for').

enumerate([X|Conditions],L):-
        prolog(cp_transform(Conditions,Goals))|
        eager_enumerate([X|Goals],L).

enumerate([X|Conditions]):-
        prolog(cp_transform(Conditions,Goals))|
        eager_enumerate([X|Goals]).

%-----------------------------------------------------

cp_transform((P holds_for X),eval(P,X)).
cp_transform(((P holds_for X),C),(eval(P,X),G)) :-
        cp_transform(C,G).
cp_transform((C1,C2),(C1,G)):-
        cp_transform(C2,G).
cp_transform(C,C).


eval(P,X):-
        (P holds_for X),Z=..[P,X],demo(ax,Z).

%-----------------------------------------------------


eager_enumerate([X|P]):-
        prolog(assert((e(X):-P)))&
        pr(e(X))&
        prolog(retract((e(X):-P))).


pr(A):-
        prove((A,k_write(A),fail))|true.
pr(_):-prolog(write(end)).


prove(true).
prove(A):-systemp(A,A1)|A1.
prove((true,B)):-
        prove(B).
prove(((A,B),C)):-
        prove((A,B,C)).
prove((k_write(A),B)):-
        prolog((A=..[c,X],nonvar(X),write(X),nl))|prove(B).
prove((A,B)):-
        systemp(A,A1)|A1&prove(B).
prove((A,B)):-
        cpsystem(A,A1)|prolog(A1)&prove(B).
prove((A,B)):-
        cpclauses(A,Clauses)|
        try_each(Clauses,A,B).

try_each([(A:-B)|_],A,C):-
        prove((B,C))|true.
try_each([_|Clauses],A,C):-
        try_each(Clauses,A,C)|true.

%-----------------------------------------------------

:- public systemp/2.
:- mode systemp(+,-).

systemp((X\=Y),prolog((X\=Y))).
systemp((X is Y),prolog((X is Y))).
systemp((X < Y),prolog((X<Y))).
systemp((X > Y),prolog((X>Y))).
systemp((X mod Y),prolog((X mod Y))).
systemp((X == Y),prolog((X==Y))).
systemp((X \==Y),prolog((X\==Y))).
systemp((X=Y),prolog((X=Y))).
systemp((X=..[Y,_]),prolog((X=..[Y,_]))).
systemp(print(X),prolog((print(X)))).
systemp(write(X),prolog((write(X)))).
systemp(nl,prolog((nl))).
systemp(not(P),prolog((\+(P)))).
systemp(demo(X,Y),prolog(demo(X,Y))).
```

[Knowledge Base]

```
ax((child((jim,mary))<--true)).
ax((child((tom,mary))<--true)).
ax((child((mary,nancy))<--true)).
ax((child((john,nancy))<--true)).
ax((child((barbara,john))<--true)).
ax((likes((tom,barbara))<--true)).
ax((likes((mary,jim))<--true)).
ax((likes((jim,nancy))<--true)).
ax((likes((tom,mary))<--true)).
ax((poorer((tom,mary))<--true)).
ax((poorer((barbara,mary))<--true)).
ax((poorer((mary,jim))<--true)).

ax((parent((X,Y))<--child((Y,X)))).
ax((ancestor((X,Y))<--parent((X,Y)))).
ax((ancestor((X,Y))<--parent((X,Z)),ancestor((Z,Y)))).

ax((brother((X,Y))<--parent((Z,X)),parent((Z,Y)),not(identity((X,Y))))).
ax((cousin((X,Y))<--parent((Z,X)),parent((W,Y)),brother((Z,W)))).


ax((richer((X,Y))<--poorer((Y,X)))).
ax((richer((X,Y))<--poorer((Z,X)),richer((Z,Y)))).

ax((identity((X,Y))<--X==Y)).

%-----------------------------------------------------


child holds_for (jim,mary).
child holds_for (tom,mary).
child holds_for (mary,nancy).
child holds_for (john,nancy).
child holds_for (barbara,john).
likes holds_for (tom,barbara).
likes holds_for (mary,jim).
likes holds_for (jim,nancy).
likes holds_for (tom,mary).
poorer holds_for (tom,mary).
poorer holds_for (barbara,mary).
poorer holds_for (mary,jim).

parent holds_for (X,Y):-child holds_for (Y,X).
ancestor holds_for (X,Y):-parent holds_for (X,Y).
ancestor holds_for (X,Y):-parent holds_for (X,Z),ancestor holds_for (Z,Y).

brother holds_for (X,Y):-parent holds_for (Z,X),parent holds_for (Z,Y),
                                    not(identity holds_for (X,Y)).
cousin holds_for (X,Y):-parent holds_for (Z,X),parent holds_for (W,Y),
                                    brother holds_for (Z,W).

richer holds_for (X,Y):-poorer holds_for (Y,X).
richer holds_for (X,Y):-poorer holds_for (Z,X),richer holds_for (Z,Y).

identity holds_for (X,Y):- X==Y.

%-----------------------------------------------------

% [Application examples]

% (1) Cross reference :

cross_ref(W,A,L):-
        enumerate([M|clause_body(W,A,M)],L).


clause_body(W,A,P):-
        Q=..[A,_],
        Y=..[W,(P1<--X)],Y,
        P1=..[P,_],
        member_term(Q,X).


% (2) Higher order sets :

second_order(L):-
        enumerate([Z|X^(enumerate([P|P holds_for X],Z),length(Z,2))],L).

%-----------------------------------------------------


member_term(X,(X,Y)).
member_term(X,(Y,L)):-member_term(X,L).
member_term(X,X).
```

[Execution Examples]

```
Prolog-20  version 1.0
Copyright (C) 1981, 1983 by D. Warren, F. Pereira and L. Byrd

| ?- [-test].

test reconsulted   1811 words      1.31 sec.

yes
| ?- enumerate({X|parent holds_for X},L).

L = [(john,barbara),(mary,jim),(mary,tom),(nancy,john),(nancy,mary)],
X = _29

yes
| ?- enumerate({X|brother holds_for X},L).

L = [(jim,tom),(john,mary),(mary,john),(tom,jim)],
X = _29

yes
| ?- enumerate({X|cousin holds_for (X,barbara)},L).

L = [jim,tom],
X = _29

yes
| ?- enumerate({P|P holds_for (tom,mary)},L).

L = [child,likes,poorer],
P = _29

yes
| ?- enumerate({P|P holds_for (nancy,tom)},L).

L = [ancestor],
P = _29

yes
| ?- enumerate({P|P holds_for (mary,tom)},L).

L = [ancestor,parent,richer],
P = _29

yes
| ?- enumerate({P|P holds_for (jim,barbara)},L).

L = [cousin,richer],
P = _29

yes
| ?- cross_ref(ax,parent,L).

L = [ancestor,brother,cousin]

yes
| ?- second_order(L).

L = [[ancestor,parent],[brother,richer],[child,richer],[cousin,likes],
     [cousin,richer]]
```

```
Concurrent Prolog version 1.0    (C) 1983 Ehud Shapiro

yes
| ?- [-testc].

testc reconsulted   2068 words      1.20 sec.

yes
| ?- cp enumerate({X|parent holds_for X}).
(mary,jim)
(mary,tom)
(nancy,mary)
(nancy,john)
(john,barbara)
end*** cycles: 5

X = _14
```

```
yes
| ?- cp enumerate({X|brother holds_for X}).
(jim,tom)
(tom,jim)
(mary,john)
(john,mary)
end*** cycles: 5

X = _31

yes
| ?- cp enumerate({X|cousin holds_for (X,barbara)}).
jim
tom
end*** cycles: 5

X = _31

yes
| ?- cp enumerate({P|P holds_for (tom,mary)}).
child
likes
poorer
end*** cycles: 5

P = _31

yes
| ?- cp enumerate({P|P holds_for (nancy,tom)}).
ancestor
end*** cycles: 5

P = _31

yes
| ?- cp enumerate({P|P holds_for (mary,tom)}).
parent
ancestor
richer
end*** cycles: 5

P = _31

yes
| ?- cp enumerate({P|P holds_for (jim,barbara)}).
cousin
richer
end*** cycles: 5

P = _31

yes
| ?- core     97280  (51712 lo-seg + 45568 hi-seg)
heap     34816 =  28415 in use +   6401 free
global    1449 =     16 in use +   1433 free
local     1024 =     16 in use +   1008 free
trail      511 =      0 in use +    511 free
    0.02 sec. for 1 GCs gaining 1100 words
    1.42 sec. for 79 local shifts and 113 trail shifts
   73.35 sec. runtime
```

## 4. Discussion

We have introduced the concept of set abstraction as an analogy of that of lambda abstraction, and proposed a predicate "enumerate" to count all elements of the set implied. The set abstraction comprises two common features concerning PROLOG : "set expression" and "predicate variables". In the usual sense, the set expression proposed and implemented in literature so far concerns only dealing with the first order data objects, while as we have seen, the set abstraction discussed here extends the set expression function so that it may handle even the second

order predicates. That is why we distinguished the set abstraction from the set expression in this paper. There are, in fact, several languages of PROLOG family where the predicate variables are permitted at the syntax level. As far as we know, however, none of them enables one to deal with predicate variables as data objects of abstraction or to obtain the set of attributes derivable from the axioms by deductive inference, neither.

A natural extension to the set abstraction suggests the possibility of introducing the higher order set abstraction such as the set of sets of attributes. This immediately leads to the problem of self-application. That is, in the presence of self-application, the well-known diagonal arguments bring us the Russell's paradoxes. A trivial way to avoid arising the paradoxes may be to restrict object worlds to finite sets. This will not impose so strict restrictions on the practical phase.

In this paper it has been shown that under the current environment of PROLOG language facility, one can easily achieve the set abstraction function which has the capability of dealing with the second order predicate logic.

The issue of efficient implementation should be discussed and studied, which is at present left open.

## ACKNOWLEDGEMENTS

The author would like to thank Dr.K.Furukawa, the chief of the second laboratoty, ICOT, and the members of the KL1 design task group at ICOT, for their useful discussion and suggestion. He would also like to express his gratitude to the referees for their useful comments on an earlier draft of this paper.

Last but not least, the author is very grateful to Dr.T.Kitagawa, the president of IIAS-SIS, Fujitsu limited, for warm encouragement as well as sharp advice.

## REFERENCES

Bowen,D.L., DECsystem-10 Prolog User's Manual, Department of Artificial Intelligence, University of Edinburgh, Dec.1981.

Bowen,K.A. and Kowalski,R.A., Amalgamating Language and Meta Language in Ligic Programming, in "Logic Programming"(eds. Clark and Tarnlund),Academic Press(1982).

Clark,K.L. and Gregory,S., PARLOG : A Parallel Logic Programming Language, Research Report DOC83/5,May(1983).

Clark,K.L.,McCabe, and Gregory,S., IC-Prolog language feartures, in "Logic Programming"(eds.Clark and Tarnlund), Academic Press(1982).

Furukawa,K.,Kunifuji,S.,Takeuchi,A.and Ueda,K., The conceptual specification of the Kernel Language version 1, ICOT Tech. Report(1984).

Hirakawa,H. and Chikayama,T., Eager and Lazy Enumerations in Concurrent Prolog, ICOT Tech. Memo TM-0036(1984).

Kahn,K., Pure Prolog Interpreter in Concurrent Prolog, Presentation at ICOT, 1983.

Kahn,K., A primitive for the control of logic programs,1984 International Symposium on Logic Programming, Atlantic City, NJ (1984).

Kowalski,R.A., Predicate Logic as Programming Language, Proc. IFIP-74 Congress, North-Holland pp.569-574 (1974).

Kunifuji,S., Asou,M., Sakai,K., Miyachi,T., Kitakami,H.,Yokota,H.,Yasukawa,H.andFurukawa,K. Amalgamation of object knowledge and meta knowledge in Prolog and its application, Reprint 30-1, Knowledge Engineering and Artificial Intelligence Working Group of the Inform. Process. Soc. of Japan , also ICOT Tech. Report TR-009 (in Japanese),1983.

Shapiro,E.Y.,A subset of Concurrent Prolog and its interpreter, ICOT Tech. Report TR-003(1983).

Warren,D.H.D., Higher-order Extensions to Prolog —— Are They Needed ?, D.A.I. Research pape No.154,University of Edinburgh, also 10th International Machine Intelligence Workshop,Case Western Reserve University, Cleveland, Ohio, April 1981.