ICOT Technical Report: TR-060

TR-060

A Note on the Set Abstraction in Logic Programming Language

by

Takashi Yokomori
(Fujitsu Ltd.)

April, 1984

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

A Note on the Set Abstraction in Logic Programming Language

by

Takashi YOKOMORI

International Institute for Advanced Study of Social Information Science(IIAS-SIS), Fujitsu Ltd.

April 1984

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 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, set abstractions get 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 logic. Further, the implementation programs written in PROLOG and Concurrent Prolog are given.

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 "Pure Prolog" program in which no illogical construct is allowed, while a practical PROLOG language 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 a predicate "setof" in DEC-10 Prolog([1]), "set" in PARLOG([3]), and "enumerate" in KL1([4]). The introduction of set expressions enables one to describe the set of all solutions to some goal in a program. As Warren discussed in [10], 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[10]. The purpose of this note is to discuss a possible extension to PROLOG called "set abstraction" and to demonstrate

the usefulness of the extension. Set abstractions can be regarded as an extension of set expressions in which predicate variables are allowed as data objects. Also, it may be possible to take set abstractions as a simple analogy of lambda abstractions. Thus, in this paper we take the position to distinguish "set abstractions" from "set expressions".

The concept of set abstraction is introduced, and the predicate "enumerate" is proposed in Section 2. The predicate "enumerate" considered here is an extenssion of the one introduced in [4]. Section 3 presents the implementation example for the predicate "enumerate". 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 set abstraction in a natural way. Set abstraction discussed in this paper is a simple analogy of λ-abstraction in the theory of λ-calculus. One may obtain a function from a term by means of λ-abstraction, while with the concept of set abstraction one can associate a relation implied by the term.

Let P be a term containing free occurrences of a variable x, where the prime functor of P is a predicate symbol. Then, analogous to λ-abstraction, one can define the concept of set abstraction in the following manner : Using a pair of braces {} instead of a greek letter λ 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 x satisfying the relation implied by P. As a notation, we write

{x| P }

for {}x.P.

For example, suppose 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,

$$\{\}P.\text{have\_property}(X,P)$$

is obtained.

The former, $\{x|\ \text{have\_property}(x,P)\}$ 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

$$\{P|\ \text{have\_property}(x,P)\}\ ,$$

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 set abstraction extension to PROLOG does really add something new to the language. In [10] Warren discussed the benefits of introducing the concepts of "predicate variables(or predicates as data objects)", "set expression", and " λ-expression" and concluded that predicate variables and λ-expression can be merely regarded as "syntactic sugar" and that they don't 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 abstraction considered here is concerned with two extensions to the language: predicate variables and set expression. As previously defined, set abstraction here can be taken as set expression in which the treatment of predicate variables is taken into consideration.

Suppose the following knowledge-base(KB) 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(mary,nancy).
```

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).
```

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.

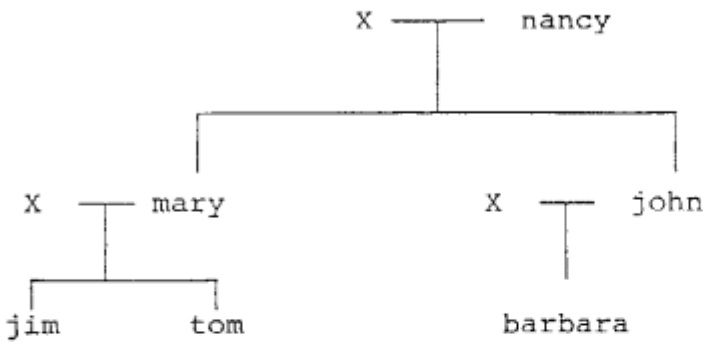

Fig.1 A family tree

Fig.1 illustrates a family relation in the KB given.

We are nowin 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 avariable. In the intensional expression, G is of the form { X | conditions } , conditions are given as a sequence of goals in Pure Prolog.

Semantics. enumerate(G,L) succeeds if and only if G is nonempty. If a set G is infinite, then L serves as a stream variable to bind elements of G. Otherwise, L does as 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 enumerate(G,L) 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 (goal) to be invoked is :

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

The answer to this query is obtained as a list:

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

In the similar way, the response 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 usage of enumerate(G,L) demonstrates the usefulness of the predicate,which distinguishs itself from other set predicates 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", 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] is obtained.

It should be remarked that there is originally no fact of the form P(mary,tom) in KB considered. Similarly, the query

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

deduces the response L = [cousin, likes]. Note that

there is no fact of relation between "tom" and "barbara" concerning " cousin ".

Thus, the predicate " enumerate " makes it possible to infer predicates(attributes) as data objects,as well as the data of first order.

Besides these higher-order 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 as a result:

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

This kind of approach to handle an infinite set has been taken in recent papers([3],[4]).

3. Implementation Examples

Two implementation programs for the predicate "enumerate" are given in this section. One is written in DEC-10 Prolog, while the other in Concurrent Prolog. 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 "enumerate" is that set predicate in conventional PROLOG is only concerned with a "finite set".

As a conclusion, it turned out that

(i) in DEC-10 Prolog implementation, the predicate "setof" is essentially required, and that the predicate "demo" plays an important role, while

(ii) it is crucial for Concurrent Prolog implementation of "enumerate" to realize the predicate "eager_enumerate" in the environment of no backtracking mechanism.

These predicates ("demo", "eager_enumerate") have been already discussed in literature([5],[8]) in reference to the Fifth Generation Computer System, and the attempt in this paper proves the usefulness of those predicates.

Notes.

(1) The predicate "enumerate" written in DEC-10 Prolog can handle only the case where the target is a finite set. The Concurrent Prolog version generates a stream

of all elements of the set involved.

(2) There are, in fact, several PROLOG languages in which predicate variables are allowed provided that they must have been instantiated at the execution time. 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 achieving the treatment of predicate variables.

(3) The predicate "enumerate" implemented here is slightly different from the one defined in the previous section in that the specification implemented allows only intentional expressions 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({P|P holds_for X}, L) :-
            var(P),!, setof(P,eval(P,X),L).
enumerate({X|P}, L) :-
            \+(P=..[holds_for| ]),!,
            setof(X,demo(ax,P),L).
```

where \+(P) is the negation of P, i.e., not(P) in DEC-10 Prolog syntax.

(in Concurrent Prolog)

```
enumerate({P|P holds_for X}, L) :-
            prolog(var(P))|
            eager_enumerate({P|eval(P,X)},L).
enumerate({X|P}, L) :-
            prolog(\+(P=..[holds_for| ]))|
            eager_enumerate({X|P}, L).
```

The predicate "demo" is an extended version of the one originally proposed by Bowen and Kowalski([2]). It has been intensively investigated and implemented by Kunifuji et al.([8]). The demo(ax,P) succeeds if a goal P succeeds in a program named "ax".

The procedure eval(P,X) defnied as follows:

```
eval(P,X) :- ax(Y),Y=..[(':-'),Z1,Z2],
             Z1=..[P,X], demo(ax,Z1).
```

commits its evaluation to the "demo" predicate.

The predicate "eager_enumerate" plays a central role in the Concurrent Prolog implementation. It has been implemented by Hirakawa and Chikayama([5]) applying the AND-parallel mechanism of Concurrent Prolog to the OR-parallel execution in Pure Prolog. There is another way of implementing "eager_enumerate" proposed by Kahn([6]) in which the OR-parallel mechanisim executes OR-clauses of Pure Prolog program in parallel. The Kahn's implementation is presented here simply because of its simplicity.

DEC-10 Prolog program :

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

enumerate({P|P holds_for X},L):-
        var(P),!,setof(P,eval(P,X),L).
enumerate({X|P},L):-
        \+(P=..[holds_for| ]),!,
        setof(X,demo(ax,P),L).

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

Concurrent Prolog program :

1)"eager_enumerate" based on that of [5]

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

enumerate({P|P holds_for X},L):-
        prolog(var(P))|
        eager_enumerate({P|eval(P,X)},L).
enumerate({X|P},L):-
        prolog(\+(P=..[holds_for| ]))|
        eager_enumerate({X|P},L).

eval(P,X):-
        ax(Y),Y=..[(':-'),Z1,Z2],
        Z1=..[P,X],demo(ax,Z1).
```

2)"eager_enumerate" based on that of [6]

```
enumerate({P|P holds_for X},L):-
        prolog(var(P))|
        eager_enumerate({P|eval(P,X)}).
enumerate({X|P},L):-
        prolog(\+(P=..[holds_for| ]))|
        eager_enumerate({X|P}).

eager_enumerate({X|Goals}):-
        prolog(assert((e(X):-Goals)))&
        pr(e(X))&
        prolog(retract((e(X):-Goals))).
```

```
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=..[e,X],nonvar(X),write(X),nl))|prove(B).
prove((k_write(A),B)):-
        prolog((A=..[e,X],var(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(print(X),prolog((print(X)))).
systemp(write(X),prolog((write(X)))).
systemp(nl,prolog((nl))).


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((X=-Y)).
```

```
%-----------------------------------------------------------

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((mary,nancy)):-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)).
```

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


| ?- [-test].

test reconsulted   886 words      0.54 sec.

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


L = [(john,barbara),(mary,jim),(mary,tom),(nancy,john),(nancy,mary)],
X = _31

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


L = [(mary,john),(john,mary),(jm,tom),(tom,jim)],
X = _31


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


L = [jim,tom],
X = _31

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


L = [child,likes,poorer],
P = _31

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


L = [ancestor,richer],
P = _31

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


L = [ancestor,parent],
P = _31

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


L = [cousin,likes],
P = _31

yes
| ?- core     91648  (46080 lo-seg + 45568 hi-seg)
heap     29184 =  27221 in use +   1963 free
global    1449 =     16 in use +   1433 free
local     1024 =     16 in use +   1008 free
trail      511 =      0 in use +    511 free
    0.00 sec. for 6 trail shifts
    5.21 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 variable". 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 so that it may handle even the second order predicates. That is why we distinguished the set abstraction from the set expression. There are, in fact, some 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 derived from the axioms by inference, neither.

A natural extension to the set abstraction suggests the possibility of introducing the higher-order set abstraction such as the set of the set 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 practical phase.

In this paper it has been shown that under the current environment of PROLOG language facility, one can easily implement the set abstraction function which has the capability of dealing with the second order predicate logic. The problem of efficient implementation is left open.

Acknowledgements

The author would like to thank Dr.K.Furukawa, the chief of Second Laboratory,ICOT, for useful discussion and suggestion. He would also like to express his gratitude to the members of the KL1 Design Task Group at ICOT for thier valuable and stimulative discussion.

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

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

[2] K.A.Bowen and R.A.Kowalski: Amalgamating Language and Meta Language in Logic Programming, Tech.Rep.of School of Computer and Inf. Sciences,University of Syracuse(1981).

[3] K.L.Clark and S.Gregory:PARLOG:A parallel Logic Programming Language,Research Rep.DOC83/5 May(1983).

[4] K.Furukawa,S.Kunifuji,A.Takeuchi and K.Ueda: The Conceptual Specification of the Kernel Language Version 1, ICOT Research Rep.(1984).

[5] H.Hirakawa and T.Chikayama: Eager and Lazy Enumerations in Concurrent Prolog, ICOT TM-0036(1984).

[6] K.Kahn : Pure Prolog Interpreter in Concurrent Prolog, Presentation at ICOT,1983.

[7] R.A.Kowalski: Predicate Logic as Programming Language, Research Memo No.70,Imperial College(1973).

[8] S.Kunifuji,M.Asou,K.Sakai,T.Miyachi,H.Kitakami,H.Yokota, H.Yasukawa and K.Furukawa: Amalgamation of Object knowledge and Meta knowledge in Prolog and its applications, Inf.Processing Society of Japan, Research Committee Material(in Japanese)1983.

[9] E.Y.Shapiro: A Subset of Concurrent Prolog and Its Interpreter,ICOT Tech. Rep. TR-003(1983).

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