A NOTE OF THE SET ABSTRACTION IN LOGIC PROGRAMMING LANGUAGE

Takashi Yokonori

International Institute 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 implementation 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 DCS-10 Prolog (Bowen 1981), "set" in PARLOG (Clark and Gregory 1983), "collect" in IM-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 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 containing 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

\[
\{ x \mid P \}
\]

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

\[
\{ x \mid P \}
\]

for \( \{ x \mid P \} \).

Suppose that, for example, a term

\[
\text{have_property}(x, P)
\]

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

\[
\{ x \mid \text{have_property}(x, P) \}
\]

Or, if \( P \) is taken for the object of abstraction,

\[
\{ \text{P} \mid \text{have_property}(x, P) \}
\]

is obtained.

The former, \( \{ x \mid \text{have_property}(x, P) \} \), 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

\[
\{ \text{P} \mid \text{have_property}(x, P) \}
\]

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.

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.

\[
\text{Suppose the following knowledge base is given:}
\]

\[
(1) \quad \text{child(jim, mary).}
\]

\[
\text{child(tom, mary).}
\]

\[
\text{child(mary, nancy).}
\]

\[
\text{child(barbara, john).}
\]

\[
\text{likes(tom, barbara).}
\]

\[
\text{likes(mary, jim).}
\]

\[
\text{likes(jim, nancy).}
\]

\[
\text{poorer(tom, mary).}
\]

\[
\text{poorer(barbara, mary).}
\]

\[
\text{poorer(mary, jim).}
\]

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

\[
(2) \quad \text{parent}(X, Y) \leftarrow \text{child}(Y, X).
\]

\[
\text{ancestor}(X, Y) \leftarrow \text{parent}(X, Y).
\]

\[
\text{ancestor}(X, Y) \leftarrow \text{parent}(X, Z), \text{ancestor}(Z, Y).
\]

\[
\text{brother}(X, Y) \leftarrow \text{parent}(Z, X), \text{parent}(Z, Y), \text{not}(\text{identical}(X, Y)).
\]

\[
\text{cousin}(X, Y) \leftarrow \text{parent}(Z, X), \text{parent}(W, Y), \text{brother}(Z, W).
\]

\[
\text{richer}(X, Y) \leftarrow \text{poorer}(Y, X).
\]

\[
\text{richer}(X, Y) \leftarrow \text{poorer}(Z, X), \text{richer}(Z, Y).
\]

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

\[
\text{enumerate}(G, L)
\]

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

**Semantics.** \(\text{enumerate}(G, L)\) 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:

\[\text{- enumerate}(\{X, Y\mid \text{parent}(X, Y)\}, L)\].

The answer to this question is obtained as a list:

\[L = [(\text{John}, \text{Barbara}), (\text{Mary}, \text{Jim}), (\text{Mary}, \text{Tom}), (\text{Nancy}, \text{John}), (\text{Nancy}, \text{Mary})].\]

In the similar way, the response to the question

\[\text{- enumerate}(\{X \mid \text{cousin}(X, \text{Barbara})\}, L)\]

will be

\[L = [\text{Jim}, \text{Tom}].\]

That is, it is seen that "Jim" and "Tom" are cousins of "Barbara".

Another type of the usage of \(\text{enumerate}(G, L)\) 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

\[\text{- enumerate}(\{P \mid P(\text{Tom}, \text{Mary})\}, L)\]

One will get the response

\[L = [\text{child, likes, poorer}].\]

Furthermore, for the query

\[\text{- enumerate}(\{P \mid P(\text{Mary}, \text{Tom})\}, L)\]

the response

\[L = [\text{ancestor, parent, richer}]\]

is obtained. It should be remarked that there is originally no fact of the form \(P(\text{Mary}, \text{Tom})\) in the knowledge base at issue. Furthermore, the query

\[\text{- enumerate}(\{P \mid P(\text{Jim}, \text{Barbara})\}, L)\]

brings the response

\[L = [\text{cousin, richer}]\]

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 makes a query

\[\text{- enumerate}(\{X \mid \text{prime}(X)\}, L)\]

then one can obtain an infinite stream of prime numbers:

\[2, 3, 5, 7, 11, 13, \ldots\]

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 such later (see Section 3).

(1) **Cross reference**

Consider the following program:

\[
\text{cross_ref}(\text{Program, Attribute, L}) :-
\text{enumerate}(\{P \mid \text{clause_body}(\text{Program, Attribute, P})\}, L).
\]

where \(\text{clause_body}(\text{Prog, A, Pred}) :-\)

\[Q \leftarrow [\text{Pred}_{1}], X \leftarrow [\text{Proc}, (P' \leftarrow X)], Y, \]

\[P' \leftarrow [\text{Pred}_{2}]\], member_term(Q, X).\]

The predicate \(\text{cross_ref}(P, A, L)\) 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 \( L \mid L \) 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:

\[
?-\text{enumerate}(L \mid \text{enumerate}([P,P(X)],L), \text{length}(L,2)),S).
\]

\[
S = \{\text{ancestor,parent},\{\text{brother,richer},\text{child,richer}\}\text{cousin,likes},\{\text{cousin,richer}\}\}.
\]

3. Implementation Examples

Two implementation programs for the predicate "enumerate" are given in this section. One is written in DEC-10 Prolog(Rowen 1981), while the other in Concurrent Prolog(Shapiro 1983). Both programs can run on DEC 2000 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(Kunifinji 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 involved. 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 variables are allowed provided that they must have been instantiated 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" implemented 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:

\[
\text{enumerate}(X \mid \text{Conditions}, L) :-
\text{transform}(\text{Conditions}, \text{Goals}),
\text{setof}(X,Y,\text{goals},L).
\]

where
\[
\text{transform}(\text{Conditions}, \text{Goals}) :-
\text{transform}(\text{Conditions}, \text{Goals}).
\]

\[
\text{transform}(\text{Conditions}, \text{Goals}) :-
\text{transform}(\text{Conditions}, \text{Goals}).
\]

\[
\text{transform1}((\text{P holds for } X),C),\text{eval}(P,X,G)):-
\text{var}(P),\text{transform}(C,G).
\]

\[
\text{transform}(\text{P holds for } X),C),\text{eval}(ax,G)):-
\text{atom}(P),\text{Q},\text{eval}(P,X),
\text{transform}(C,G).
\]

\[
\text{transform1}((\text{C}),G) :-
\text{transform}(C,G).
\]

\[
\text{transform}(\text{P holds for } X),\text{eval}(P,X)) :-
\text{var}(P).
\]

\[
\text{transform}(\text{P holds for } X),\text{eval}(ax,G)) :-
\text{atom}(P),\text{Q},\text{eval}(P,X).
\]

\[
\text{transform}(C,G).
\]

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 "axm" is an extended version of the one originally proposed by Bowen and Kowalski (Bowen and Kowalski 1982). It has been intensively investigated by Kunifaji et al. (Kunifaji et al. 1983). The predicate \texttt{demo(axm,F)} succeeds if a goal \( F \) succeeds in a program named "axm".

The procedure \texttt{eval}(P,X) defined by

\begin{align*}
\texttt{eval}(P,X) & : \texttt{axm}(X) \\
& \quad \text{eval}(P, X, Z = \langle Z_1 \rangle, Z_2 = \langle P, X \rangle), \texttt{demo}(axm, Z)
\end{align*}

commits its evaluation to the predicate \texttt{demo}.

( in Concurrent Prolog )

\begin{align*}
\texttt{enumerate}(X, \text{Conditions}, \text{Goals}) & : \\
& \quad \texttt{prolog}([\text{Conditions}, \text{Goals}]) || \\
& \quad \texttt{eager\_enumerate}(X, \text{Goals}), \text{List}.
\end{align*}

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

\begin{align*}
\texttt{eval}(P,X) & : \\
& \quad P \texttt{ holds\_for} X, Z = \langle P, X \rangle, \texttt{demo}(axm, Z).
\end{align*}

The predicate "eager\_enumerate" plays a central role in the Concurrent Prolog implementation. The predicate \texttt{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 \texttt{eager\_enumerate} function proposed by Kohn (Kohn 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 "axm", the Concurrent Prolog implementation requires another copy of the knowledge base in which a representation "P holds\_for X" is used for "P(X)".
(Concurrent Prolog Implementation)

:- use_module(library(clause)).

enumarate([X in Conditions, L0]):-
prolog(system([X in Conditions, L0])).

enumarate([X in Conditions, L0], L1):-
prolog(system([X in Conditions, L0, L1])).

eval(X, 0, 0).

eval(X, X, 1).

eval(X, Y, L):-
X in Conditions, L1, L2,
prolog(system([X in Conditions, L1, L2])).

(Prolog implementation for concurrent evaluation)

-\( \text{enumarate}([X \in \text{Conditions}, L0]) \equiv \text{eval}(X, 0, 0) \)
-\( \text{enumarate}([X \in \text{Conditions}, L0], L1) \equiv \text{eval}(X, X, 1) \)
-\( \text{eval}(X, Y, L) \equiv \text{eval}(X, Y, L1) \)
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. Purukawa, the chief of the second laboratory, ICOT, and the members of the KL 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 TIAS-SIS, Fujitsu limited, for warm encouragement as well as sharp advice.

REFERENCES


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


Warren, D.R.D., 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.