記 号 処 理 17－4
(1982. 1. 22. 23)

Comments on PROLOG from the function class point's view

Toshiaki Kurokawa

Information Systems Laboratory, TOSHIBA R&D Center, Kawasaki, 210, JAPAN

INTRODUCTION

In this paper, we consider about a "logic programming language" which has suitable properties in developing very large practical artificial intelligence programs.

First, we will briefly review the background of logic programming in AI. Although there are disputes between those who promote logic and those who advocate experience accumulation, it is widely recognized that a programming tool which can utilize the logical feature is necessary for the further pursuit of AI experiments. Even if one claims that the logic is useless and the accumulation of specialists' knowledge is vital for the knowledge engineering program, one needs some logic to organize the obtained knowledge.

To summarize the debate, it is helpful to read both Winograd's paper[5] and Hayes' critics[6]. Hayes precisely explained that logic is not a programming system itself, instead it underlies any programming system. He also criticized Winograd's view of the accumulation of independent logical piece and single determined logical meaning. From Hayes' point of view, there would be strong interactions between logical clauses and that the meaning of the logical sentence is more abundant and more complex than that of the instruction of procedural languages such as Lisp.

Today, PROLOG[1] would be the best-known "logic programming language" for AI research. It has been developed and used mainly in Europe for several years. Recently, United States and Japan started to accumulate the experience on PROLOG.[7,8,9]

From ours, however, it is known that PROLOG has several problems for our purposes.

PROBLEMS ON PROLOG

In the following discussions, we use Edinburgh version of PROLOG as the typical PROLOG language[2], mainly because the author's experience is limited to it and partly because plenty of information is available on it. We miss that the origial PROLOG developed at Marseille lacks for these information.

From our point of view, PROLOG has the following problems:

1) Cut operator is a too primitive and too dangerous operator to avoid the infinite loop or to achieve efficiency. Further, it implicitly destroys the two-way, equational property of the logical program.

2) Backtracking is, as is well known in the 70's, not a good tool to realize non-deterministic

search. The arguments against the late micro-PLANNER[3] are applicable to the present PROLOG programming.

3) Integer arithmetic operation, such as +, -, *, /, breaks the harmony of the simple and beautiful unification scheme in PROLOG. The novices are perplexed when he encounters unexpected results owing to the arithmetic operations. Moreover, the arithmetic primitives cannot be defined in the PROLOG programs, although other primitives are user definable.

4) PROLOG lacks for the well-known AI primitives such as context-switching, pattern-directed process invocation, controlled pattern matching, etc. Although they can be deliberately incorporated by users, it is not a very easy task.

NEW WAY TO ...

Considering above problems, we must find a new language for Knowledge Utilization and Representation Oriented LOGical programming which has the following specialities:

1) Choice of the basic unification strategy.

2) High-level primitive covering wider range including sequential/parallel processing.

3) Introducing a new value based upon the open world assumption.

4) Data abstraction and argument evaluation through the function class.

5) Introducing user-accessible short-term-memory to support very high-level controls.

ELIMINATION OF CUT OPERATOR

Cut operator, denoting "!", is used for the following purposes in PROLOG:

1) To realize the primitive not, i.e.

```
not(X) :- X,!,false.
not(X).
```

2) To implement if_then_else clause. In the special case, this is to avoid the infinite loop. An example is given below:

```
if_then_else(P,Q,R) :- P,!,Q.
if_then_else(P,Q,R) :- R.
```

3) For the efficiency's sake, i.e. (i) elimination of useless backtracking in sequential operation. (ii) elimination of information space for the possible backtracking. For example,

```
d(U+V,X,DU+DV) :- !, d(U,X,DU), d(V,X,DV).
d(U-V,X,DU-DV) :- !, d(U,X,DU), d(V,X,DV).
d(U*V,X,DU*V+U*DV) :- !, d(U,X,DU), d(V,X,DV).
```

```
d(U^N,X,N*U^N1*DU) :- !, integer(N), N1 is N-1, d(U,X,DU).
d(-V,X,-DV) :- !, d(V,X,DV).
d(exp(V),X,exp(V)*DV) :- !, d(V,X,DV).
d(log(V),X,DV/V) :- !, d(V,X,DV).
d(X,X,1) :- !.
d(C,X,0) :- atomic(C), C \== X, !.
```

is the differential program, where cut-operator, !, is used to eliminate further back-tracking useless in the format selection.

In the new language, however, not-operator will be a system primitive over the Boolean value, true or false. not(unknown) is defined unknown. There is another primitive, undefined, to check the unknown-value. undefined (unknown) is true, and undefined (true/false) is false.

If_then_else is included in the special case of select primitive such as select[Q;R] when[P;not(P)] or select[Q;R] when[P;undefined(P)].

The reader should note that the if_then_else defined by the cut operator is ambiguous; using the new language's select primitive, there are two different versions, i.e. when [P;not(P)] and when [P;undefined(P)]. To be more precise, the definition should be select[Q;R;S] when [P;not(P);undefined(P)].

As for the efficiency, such primitives as select, sequence, iterate, will bring the efficient coding in the new language. For example, the differential program above can be written as the following:

```
d(Form,X,Answer) :-
      select [Answer = DU+DV; Answer = DU-DV; Answer = DU*V+U*DV;
              Answer = -DV; Answer = exp(V)*DV; Answer = DV/V;
              Answer = 1; Answer = 0]
      when [Form = U+V; Form = U-V; Form = U*V; Form = U^N;
            Form = -V; Form = exp(V); Form = log(V); Form = X;
            Form = C] <d(U,X,DU),  d(V,X,DV), integer(N),
                       N1 is N-1, atomic(C), C \== X>.
```

The concise and more readable notation can be given below:

```
d(Form,X,Answer) :-
  [if
    || Form = U+V ---> d(U,X,DU),  d(V,X,DV), Answer = DU+DV;
    || Form = U-V ---> d(U,X,DU),  d(V,X,DV), Answer = DU-DV;
    || Form = U*V ---> d(U,X,DU),  d(V,X,DV), Answer = DU*V+U*DV;
    || Form = U^N ---> d(U,X,DU), integer(N), N1 is N-1,
```

```
                    Answer = N*U^N1*DU;
    ;; Form = -V ---> d(V,X,DV), Answer = -DV;
    ;; Form = exp(V) ---> d(V,X,DV), Answer = exp(V)*DV;
    ;; Form = log(V) ---> d(V,X,DV), Answer = DV/V;
    ;; Form = X ---> Answer = 1;
    ;; Form = C ---> atomic(C), C *== X, Answer = 0;
    fi].
```

STRUCTURE OF STM

STM (short term memory) is a working environment in the new language. Whenever a clause is activated, the generated instance has a STM where the following elements are contained:

1) initial condition (initial environment)

To activate the clause or the primitive, this condition is used. Arguments of the clause are included here as well as the top-level environment.

2) final results (final value)

After execution (unification), the results are set. In case of arithmetic operation, for example, the computed value is set here.

3) history of the execution

Intermediate values are stocked here. The candidates for the non-deterministic operations are also stored here. A copy of the program counter and control sequence would be set here when the parallel processing were assumed.

4) mail-box (communication channel)

This is a media of process coordination. Through this mail-box, execution units of the new language can cooperate with others. Also the daemons set emergent order, and read the special case.

FUNCTION CLASS

The new language introduces the data type capability to the clause. The key concept is the function class[4] where a part of data abstraction is realized.

Each clause identifier employs a function class where the following elements are included:

1) Processing of arguments.

There are four options for each argument:

i) Without evaluation -- in the default case.

ii) Conditional evaluation -- when the condition is satisfied then the argument will be evaluated. For example, +(x,y) will evaluate arguments if they are composed from numeri-

cal atoms.

iii) Evaluate anyway -- do evaluation. Errors which occur during evaluation will be handled in the evaluation procedure.

iv) Evaluate with undoing information -- evaluate, but preparing the undoing when the failure occurs hereafter.

2) Data type confirmation.

This is what we call data abstraction support facility. Described here are the type information each argument should satisfy. For example in append(_1,_2,_3), _i must be a list element. In case of add(_1,_2,_3), they are numbers. The arguments of gcd(_1,_2,_3) must be natural numbers, i.e. positive integers.

Accompanying data type conditions is the returning state when the type violation occurs; it should be handled as an error or a failure.

4) Returned value.

This is the post-unification item.

i) Special value should be returned or default value, i.e. true or undefined, would be returned. This item is necessary for the functional primitive.

ii) Form of the value in case of the functional. All possible values are returned, or only one candidate is enough, or returning one and prepare others for further computation.

COMMENTARY ON PROLOG CONCERNING FUNCTION CLASS

Several comments on PROLOG relating function class are given here.

1) PROLOG is ambiguous on assignments of variables when plural choices exist.

For example, append([1|X],Y,[1,2,3]) has three possible solutions for X and Y: ([2,3], []), ([2], [3]), and ([], [2,3]). It is not easy to describe whether the user wants all, or only one, or temporally one with candidates left. Actually, it is known only after the unification, i.e. if there appears cut symbol, then only one is enough. If forcing fail comes, then all are necessary.

In PROLOG, one is returned with others stored for later use even if they are unnecessary.

2) In PROLOG, false is returned unless the matching clause exists.

However, it is ambiguous when the explicit (forcing) false occurs or the appropriate clause cannot be found. Think, for example, about the case when a careless user forget to include a clause.

If the closed world assumption does not hold, the explicit false and the failure to prove should be distinguished clearly.

3) It is ambiguous in PROLOG that the form of the argument implies a pattern or value. For

example, in d(U-V, X, DU-DV) :- ..., 'U-V' means a pattern or a structure where a '-' combines two substructure 'U' and 'V'.

In the case of d(X**N, X, N*X**(N-1)) :- ..., N-1 means a value one less than N. Actually, PROLOG avoids this ambiguity by restricting the evaluation in the special pre-defined primitives.

However, this will be a fatal weakness when one tries to extend the pattern matching

4) Function class concept is developed originally for Lisp-like functional computation where the number of formal arguments are pre-determined according to the function.

In PROLOG, however, the number of slots are fixed but the number of variables are not fixed. Variables are embedded into the input pattern without data type declaration.

RELATING WORKS

There are several works other than ours which tries to extend the some features of PROLOG, which are briefly reviewed here.

1) Prolog/KR by Nakashima[9]

Prolog/KR is a PROLOG dialect embedded in UTILISP[10]. Included are high-level deterministic sequencing primitives, data base daemons, >x type input arguments which should be evaluated to atomics, and context-switching by WITH statement.

Prolog/KR tries to incorporate realistic features into PROLOG, although somewhat in arbitrary fashion. The strategy seems to peculiar in Lisp community, i.e. let's_try_it_and_think_it_later fashion. Actually, the new language will be indebted this strategy to Prolog/KR.

2) DURAL by Goto[8]

DURAL is a simple language embedded in Maclisp on DEC-20. Its syntax depends wholly on Lisp just as Prolog/KR. DURAL employs both depth-first and unit-resolution for the basic mechanism. Its speciality is the modal operator based on the natural deduction such as LK.

3) Uniform by Kahn[13]

Uniform intends to unify Lisp, Prolog, and actor languages. It is again embedded in Lisp. The basic mechanism is, Kahn claims, augmented unification which does pattern-matching, evaluation, and message-sending.

However, it lacks for versatile control mechanisms. The usage of environments instead of pure instances brought evaluation facility just as STM in the new language. Kahn claims that the present state of the Uniform is not a perfect one, so we had better to wait for the ultimate version of Uniform, where much will be changed.

ACKNOWLEDGEMENT

The author would like to acknowledge the members of the logic programming committee for the Fifth Generation Computer Systems, especially Mr. Yokoi at ETL, Mr. Goto at ECL. Special thanks are also due to Mr. Takei at TOSHIBA R&D for his helpful discussions. Mr. Maeda kindly helped in enhancing the word processing facility for this document.

REFERENCES

[1] Kowalski, R.,"Algorithm=Logic+Control",CACM,22,7,424-436,(Jul. 1979).

[2] Pereira L. M., F.C.N. Pereira, and D.H.D. Warren,"USER'S GUIDE to DECsystem-10 PROLOG", 60, University of Edingburgh,(Sep. 1978).

[3] Sussman, G.J.,"Why Conniving is better than Planning", MIT AI Memo 255, (Feb. 1972).

[4] Kurokawa, T., "The Function Class", Conference Record of the 1980 LISP Conference, 38-45, (1980).

[5] Hayes, P. J.,"In Defence of Logic", IJCAI 77, 559-565 (1977)

[6] Winograd, T., "Frames and the declarative-procedural controversy",in Representation and Understanding, Academic Press, (1975)

[7] Sowa, J. F., "A PROLOG TO PROLOG", IBM Systems Research Institute,32,(Jan. 1981)

[8] Goto, S.,"jutsugo-gata programming gengo DURAL to sono shori-kei",ECL Report 16892, 46, (May 1981).

[9] Nakashima, H., "Prolog/kr", Univ. of Tokyo, Memo, (1981)

[10] Chikayama, T., "UTI-LISP Manual", Univ. of Tokyo, Memo, (1981)

[11] Andrews, P.B., "Theorem Proving via General Matings", JACM, 28, 2, 193-214, (Apr. 1981)

[12] Proc. of International Conference on Fifth Generation Computer Systems, Oct. 19-22, from JIPDEC JAPAN, (1981)

[13] Kahn, K. M., "Uniform -- A Language based upon Unification which unifies (much of) Lisp, Prolog, and Act I", Proc. 7th IJCAI, (Aug. 1981)