New Generation Computing, 4 (1986) 97-105
OHMSHA, LTD. and Springer-Verlag

Short Notes

Translating Production Rules into a Forward Reasoning Prolog Program

Akira YAMAMOTO* and Hozumi TANAKA
Department of Computer Science,
Tokyo Institute of Technology,
2-12-1, Ookayama, Meguro-ku, Tokyo 152, Japan.

Received 1 April 1985

Abstract Several attempts have been made to design a production system using Prolog. To construct a forward reasoning system, the rule interpreter is often written in Prolog, but its execution is slow.

To develop an efficient production system, we propose a rule translation method where production rules are translated into a Prolog program and forward reasoning is done by the translated program. To translate the rules, we adopted the technique developed in BUP, the bottom-up parsing system in Prolog.

Man-machine dialogue functions were added to the production system and showed the potential of our method to be applied to expert systems.

Keywords: Production System, Expert System, Prolog.

§1 Introduction

Production systems are widely used in many expert systems, one example of which is MYCIN.[1] In most cases, these production systems have been implemented using Lisp. Recently, Prolog has been receiving a great deal of attention as a language suitable for artificial intelligence, and there have been some attempts to develop a production system using Prolog.

There are two ways of reasoning in a production system; forward reasoning and backward reasoning. The backward reasoning system can be easily realized using Prolog.[2] Since Prolog executes its programs in a backward manner, it is a simple matter to make use of this mechanism for implementing backward reasoning systems. For forward reasoning systems, however, we cannot directly use Prolog's executing mechanism; some other method, such as a rule interpreter written in Prolog is necessary to realize a forward reasoning system.[3,4]

In the rule interpreter method, production rules written by a user are stored in the database as they are, and the rule interpreter reads and interprets them at the time

* Present address: Reservoir modelling department, Nippon Schlumberger K. K., 2-2-1, Fuchinobe, Sagamihara-shi, Kanagawa 229, Japan.

of execution. This method can be implemented simply to make a flexible system, but the inference speed is slow.

As an alternative, we propose a rule translation method, in which each production rule is translated into a Prolog clause; by executing this Prolog program, forward reasoning can be done without a rule interpreter. As the Prolog program obtained by the translation runs directly, the reasoning speed is faster than with the rule interpreter method. Furthermore, the obtained program can be compiled for speed-up.

To produce a Prolog program from production rules, we adopted the translation technique of BUP,[5] the bottom-up parsing system in Prolog in which grammar rules are translated into a Prolog program. Therefore the system quickly finds the next applicable rule without having to search all the translated rules, and the top-down prediction mechanism reduces the number of unnecessary rule trials.

Section 2 describes our translation-method and shows how the translated program works.

Section 3 presents an explanation of the man-machine interactive dialogue functions which we added to this production system in order to make interactive reasoning possible.

§2 Translation-method Forward Reasoning Production System

2.1 Translation of the Production Rule Written in Propositional Logic

In our translation-method, production rules are written in the Horn clause of first-order predicate logic; the translator then translates them into a Prolog program.

First we will explain the translation of production rules written in propositional logic.

A production rule is written in the following form:

$$c <= c_1, c_2, \ldots, c_n. \qquad (n \geq 1)$$

It is then translated into the following Prolog clause:

$$c_1(G) :- \text{link}(c,G), \text{goal}(c_2), \ldots, \text{goal}(c_n), c(G).$$

When c_i $(2 \leq i \leq n)$ is the negative proposition "not c_i'", "not goal(c_i')" is generated instead of "goal(c_i)". This Prolog clause acts as follows: when c_1 is inferred, try proving c_2 - c_n; if the proof succeeds, call the consequent c. Here, G is either the final consequent of the inference or the top-down prediction goal.

A known fact is written as

c.

and it is translated into the form

fact(c).

In addition to translating rules and facts into Prolog clauses, the translator also generates a goal clause, link clauses, and terminate clauses.

A goal clause is defined as follows:

goal(G) :- fact(H), link(H,G), P = ..[H,G], call(P).

It picks up one known fact H and calls a rule clause having H as a head.

Link clauses indicate proposition pairs, in which one of the propositions has the possibility of being proven by the inference starting with the other proposition. That is to say, for a rule

$$c <= c_1, c_2, ..., c_n.$$

if c_1 is proved, there is a possibility that c will also be proved. Thus we have a link clause

$$link(c_1, c).$$

A link clause is made for each production rule. Furthermore, we define the link relation as a transitive and reflective relation. Accordingly, for any proposition pair (x, z) which with some proposition y has the relation

link(x, y)   and   link(y, z)

the translator generates

link(x, z).

and it also generates

link(X, X).

We let the link check always succeed when the second argument G has no value, so the following clause is added at the top of the link clauses:

link(_, G) :- var(G), !.

A terminate clause is made for a proposition c as

c(c).

Terminate clauses are made for any proposition which appears either in the premise part or in the consequent of any rule. Terminate clauses have the function of stopping the inference by instantiating the variable goal G with c or by detecting the coincidence of the top-down prediction goal G with c.

2.2 Execution of the Program Obtained by Translation

Suppose the knowledge base contains the rule

$$c <= c_1, c_2, c_3. \ .$$

This is translated as

$$c_1(G) :- link(c, G), goal(c_2), goal(c_3), c(G).$$

Forward reasoning is started by calling "goal(G)". The goal clause picks up a known fact and begins the inference with it. The link check will succeed when G is a variable. Suppose we have the fact c_1; the goal clause calls a clause having c_1 as its head. The above rule clause then checks whether the conditions c_2 and c_3 hold using the goal clause, and if the check succeeds, the consequent c is called. At this point, if the terminate clause of c is selected, the inference stops by instantiating G with c. Or if there is a rule clause with the head c, the inference can be continued by calling this clause.

When the goal clause is called with a G which has a top-down prediction goal as its value, the system will try to prove it by forward reasoning. In this case, link relations are used to guide the inference to that goal. The link predicate in the goal clause checks whether there is a possibility to conclude the given goal by starting the inference with the picked-up fact. The link predicate in the rule clause checks the same possibility with the consequent of the rule. If the link check fails, another fact or rule is selected by backtracking. When an inferred consequent coincides with the given goal, a terminate clause is used and the proof of the top-down prediction goal succeeds.

2.3 Example of Translation

A sample of production rules and facts and the result of their translation is shown in Fig. 1.

```
production rules and facts

watermelon <= produce, 1bs15.
produce    <= green.

green.
1bs15.
```

↓ translation

```
Prolog program for forward reasoning

% goal clause
goal(G) :- fact(H), link(H, G), P =..[H, G], call(P).

% terminate clauses
green(green).
1bs15(1bs15).
produce(produce).
watermelon(watermelon).

% rule clauses
produce(G) :- link(watermelon, G), goal(1bs15), watermelon(G).
green(G) :- link(produce, G), produce(G).

% link clauses
link(_, G) :- var(G), !.
link(green, produce).
link(green, watermelon).
link(produce, watermelon).
link(X, X).

% fact clauses
fact(green).
fact(1bs15).
```

Fig. 1 Example of translation.

The obtained program can be executed by

```
?- goal(G).
```

and we get the answer G = green, produce, watermelon, 1bs15, successively, using backtracking.

The execution flow in the case for which we get the answer G = watermelon is shown in Fig. 2. In this figure, "→" means the body expansion of the called clause, "=" indicates the equivalent literal, and "—" indicates the matching of two literals.

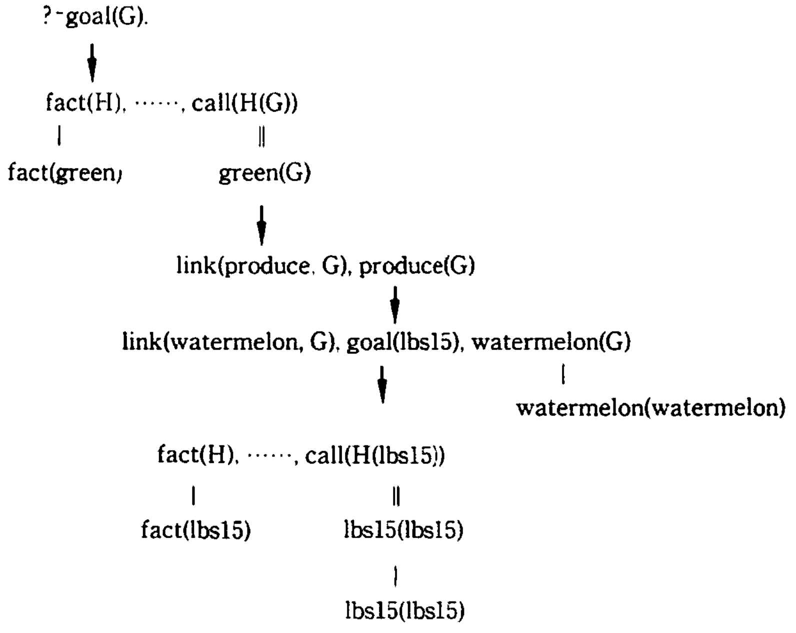
?-goal(G).
fact(H), ..., call(H(G))
fact(green)
green(G)
link(produce, G), produce(G)
link(watermelon, G), goal(lbs15), watermelon(G)
watermelon(watermelon)
fact(H), ..., call(H(lbs15))
fact(lbs15)
lbs15(lbs15)
lbs15(lbs15)

Fig. 2 Sample execution flow.

From the fact green, produce is inferred. Then the rule for watermelon is applied and it tries to prove lbs15 using the goal clause with the top-down prediction.

2.4 Translation of Production Rules Written in First-order Predicate Logic

In Section 2.1, we explained the translation of the rules of propositional logic. In this section, production rules are extended to first-order predicate logic. One difficulty is the treatment of the arguments of the predicates in the rules.

The predicate names of the first condition of the premise part and the consequent of the rule are themselves used as predicate names in the translated Prolog program. Therefore, we must consider how to handle the arguments of those predicates. We pack those arguments into a list, and this list becomes the first argument of the translated clause. Other conditions in the rule become the argument of the predicate "goal" without any change in their forms. For example, the rule

```
should_take(X, Y) <= complain(X, Z), suppress(Y, Z), not unsuitable(Y, X).
```

is translated into the following Prolog clause:

```
complain([X, Z], G) :-
        link(should_take(X, Y), G),
        goal(suppress(Y, Z)),
        not goal(unsuitable(Y, X)),
        should_take([X, Y], G).
```

Terminate clauses are generated as:

```
should_take([X, Y], should_take(X, Y)).
complain([X, Z], complain(X, Z)).
suppress([Y, Z], suppress(Y, Z)).
unsuitable([Y, X], unsuitable(Y, X)).
```

Arguments of fact clauses and link clauses become compound terms like:

```
fact(suppress(aspirin, pain)).
link(complain(X, Z), should_take(X, Y)).
```

The goal clause is changed as follows:

```
goal(G) :- fact(H), link(H, G), H = ..[PredH|ArgH],
                                P = ..[PredH, ArgH, G], call(P).
```

Notice that when a transitive link is generated, unification of the arguments occurs. For example, if

```
link(a(X), b(f(X)))      and      link(b(Y), c(g(Y)))
```

then the generated link clause is

```
link(a(X), c(g(f(X)))).
```

Only one terminate clause is made for each predicate of the same arity.

§3 Man-machine Interactive Dialogue Functions

3.1 Interactive Reasoning

When we think of potential applications of the production system, such as in expert systems, it is important that the production system be capable of working interactively with the person using the system. Among the many possible functions of man-machine interaction, we will consider the following two functions in this paper: one function consists of having the production system ask the user about the fact to be used for the inference when the relevant information is lacking in the database. The other function is to explain the reason behind such a question when the user wants to know why the system has asked him that. The second function is called the "why explanation".[2)]

It is easy to implement these functions in the interpreter method.[6)] As the execution of the system is fully controlled by the rule interpreter, we only have to add the interface program to the interpreter. On the other hand, it seems a little difficult to realize these functions in the translation-method.

We investigated whether it was necessary to include these functions in every translated rule clause, and we concluded that it was not. Fortunately, in our system any sub-part of the entire inference is started by the goal clause, so the dialogue functions can be realized simply by changing the goal clause. In the following sections, we will explain the implementation of the above two functions in our translation method production system.

3.2 Asking the User Questions

In order to achieve dialogue capability in our production system, we changed the goal clause into the following two clauses:

```
goal(G, Iex) :- fact(H), link(H, G),
        H = ..[PredH|ArgH], P = ..[PredH, ArgH, G, Iex], call(P).
goal(G, Iex) :- link(H, G), $askable(H), reported(H, Iex),
        H = ..[PredH|ArgH], P = ..[PredH, ArgH, G, Iex], call(P).
```

The first goal clause is almost the same as before. When this first goal clause fails, the second goal clause is called. Using link relations, it finds a predicate H which is either G itself or some predicate which will start the inference to prove G. If H is recorded as $askable, it asks the user about H using the program "reported". The argument Iex contains the information for explanation. When "reported" succeeds, the inference is continued in the same way as in the first goal clause.

The program "reported(H, Iex)" which we made referring Hammond,[6] asks the user if H holds or not. When the user replies yes, "reported" succeeds. When H has a free variable (for example, suppress(X, pain)), we can reply with the value of the variable (for example, aspirin). This program remembers the answers of the user and never asks the same question again.

3.3 Explanation of Reasoning behind the Question

To the question asked by "reported", the user, conversely, can ask "why" to the system. Then the system explains why it asked the first question. The information Iex is used for this explanation.

We let Iex have a list of rules which have called the goal clauses that have not yet terminated. That is to say, when a rule clause calls a goal clause, it adds to Iex the rule itself in the form of the original production rule. At this point free variables in the rule may be instantiated. With this information, when a "why question" is asked by the user, the system can indicate the rules which the system is currently manipulating. When some goal clause is finished, the rule which called it is removed from Iex.

In order to realize this explanation function, we translate the rule as in the following example, and the information Iex is passed to the goal clause:
A rule

```
should_take(X, Y) <= complain(X, Z), suppress(Y, Z), not unsuitable(Y, X).
```

translates into

```
complain([X, Z], G, Iex) :-
    link(should_take(X, Y), G),
    Rule = (should_take(X, Y) <= complain(X, Z),
                suppress(Y, Z), not unsuitable(Y, X)),
    goal(suppress(Y, Z), [Rule|Iex]),
    not goal(unsuitable(Y, X), [Rule|Iex]),
    should_take([X, Y], G, Iex).
```

3.4 Example of the Dialogue

We will next take a sample knowledge base from Hammond,[6] and show its execution in our translation-method. The knowledge base which will be used here is listed below; these rules infer a medicine Y which the man X should take when he complains of the condition Z.

```
should_take(X, Y) <= complain(X, Z), suppress(Y, Z), not unsuitable(Y, X).
```

```
unsuitable(Y, X) <= aggravate(Y, Z), condition(X, Z).

suppress(aspirin, pain).
suppress(lomotil, diarrhoea).
aggravate(aspirin, peptic_ulcer).
aggravate(lomotil, impaired_liver_function).

$askable(complain(X, Z)).
$askable(condition(X, Z)).
```

When the above rules and facts are translated, they execute as follows:

```
?- goal(should_take(john, X), []).

complain(john, _14)?: pain.

condition(john, peptic_ulcer)?: why.
because
1. If:    aggravate(aspirin, peptic_ulcer), condition(john, peptic_ulcer)
   Then:  unsuitable(aspirin, john)
2. If:    complain(john, pain), suppress(aspirin, pain),
          not unsuitable(aspirin, john)
   Then:  should_take(john, aspirin).

condition(john, peptic_ulcer)?: no.

X = aspirin
```

§4 Discussion

Through the rule interpreter method, we can easily make a production system in Prolog, but its execution speed is slow. What we have found out about writing rule interpreters is that many of the tasks of the rule interpreter are similar to those of the Prolog program interpreter. If we can impose such tasks on the Prolog interpreter, we can make an efficient production system. This can be accomplished by translating rules into Prolog programs as we have described in this paper, and by doing so we can make full use of Prolog's high power. Furthermore, in the rule translation method, the translated Prolog program can be compiled and a further speed-up can be expected.

In our translation-method, we adopted the translation technique of BUP.[5] There are two major advantages to this technique. First, a rule is translated into a Prolog clause whose head is the first condition of the premise part of the rule. Accordingly, even when there are a huge number of rules, the system can pick out the necessary rule efficiently without searching all of the rules. Secondly, the top-down prediction mechanism is introduced. In a naive forward reasoning system without this mechanism, a rule can be applied when all of the conditions of its premise part have already been proved. In this system, when the first condition of the premise part is proved, the system sets the other conditions to be subgoals and actively tries to prove them. For their proof, forward reasoning, guided by these top-down subgoals, is carried out. Because of this mechanism the system doesn't have to try the same rule again and again, waiting for all of the conditions of the premise part to be proved. In effect, backward reasoning has been added to forward reasoning in our system.

One problem in this system is that if the production rule is cyclic, the system loops infinitely. One example case is that in which a proposition c_2 is proved by a proposition c_1 and c_1 can be inferred using c_2. In the case of propositional logic, any recursion in the rules results in the infinite loop of the execution. In the case of first-order predicate logic, recursive rules are sometimes harmless. However, we have not yet solved the problem of how to determine what kinds of recursion this system can and cannot handle.

Another problem is the huge number of link clauses that are obtained when the transitive relations are enumerated at the time of translation. Although it is possible to determine the transitive relation at the time of execution, this makes the system slower.

§5 Summary

A rule translation method for a forward reasoning production system in Prolog has been developed. As the production rules are translated into the Prolog program in advance, the execution speed is fast.

Interactive reasoning is also possible in the translation-method and we have shown it by implementing the "why explanation".

References

1) Shortliffe, E. H., *Computer-Based Medical Consultation: MYCIN*, Elsevier Scientific Publishing, 1976.
2) Clark, K. L. and McCabe, F. G., "PROLOG: A Language for implementing expert systems," *Machine Intelligence, 10* (Hayes & Michie, eds.), Ellis and Horwood, pp. 455-470, 1980.
3) Mizoguchi, F., "The Design of Expert Systems Using Prolog", *Journal of Information Processing Society of Japan, Vol. 25, No. 12*, pp. 1386-1395, 1984. [in Japanese].
4) Furukawa, K., "Description of Production Systems Using Prolog," in *Knowledge Engineering* (Tanaka, K. ed.), Asakura, pp. 138-139, 1984. [in Japanese]
5) Matsumoto, Y., Tanaka, H., Hirakawa, H., Miyoshi, H. and Yasukawa, H., "BUP: A Bottom-Up Parser Embedded in Prolog", *New Generation Computing, Vol. 1, No. 2*, pp. 145-158, 1983.
6) Hammond, P., "micro-PROLOG for Expert Systems," in *micro-PROLOG: Programming in Logic* (Clark, K. L. and McCabe, F. G., eds.), Prentice-Hall International, pp. 294-319, 1984.
7) Yamamoto, A. and Tanaka, H., "A Study on Production System Using Prolog," *Proc. of the 1st Conference of Japan Society of Software Science*. pp. 5-8, 1984. [in Japanese]