PR0L0G/EX1, AN INFERENCE ENGINE WHICH EXPLAINS BOTH YES AND NO ANSWERS

Adrian Walker

IBM Research Laboratory
San Jose, California, USA

ABSTRACT

The language Prolog owes much of its increasing popularity to the fact that one can use it to write knowledge based systems in a declarative style, writing a specification which is also executable. When a system written in Prolog produces a yes answer to a question, it can be made to produce an explanation of its reasoning

However, some reasonable specifications written in Prolog do not produce any answer when executed. Also, when a knowledge base answers no to a question, it does not explain why.

This paper describes a new inference engine, called Prolog/Exl. Many programs which do not produce any answer in Prolog do produce answers in Prolog/Exl. Prolog/Exl provides explanations of both yes and no answers.

I INTRODUCTION

It has been pointed out that advice from an expert system may only be useful if the system can explain the reasons for the advice. For example, given the advice "your car cannot be repaired for less than the cost of a new one", most people would want to know why. Several existing expert systems can explain their activities to some degree (e.g. 10, 11), and Michie (5) has argued persuasively that expert systems should be understood by their users.

In many expert systems, the knowledge required for expertise is held in the form of assertions and if-then rules (7, 10). Most such systems answer a question by chaining through rules to reach assertions. In (13) it is proposed that it is useful to think of an explanation as an edited proof, and to think of a proof as an edited trace. (11) takes a similar point of view.

The assertions and rules in a system can be used to represent a portion of the real world in two different ways. Under the "closed world assumption" (8) only positive facts*are stored, and it is assumed that all facts which cannot be deduced are false. The representation is the one used in relational data bases (3). Under the "open world assumption", both positive and negative facts are present, and a fact which is not listed is considered to be unknown. For example, the Emycin (10) confidence factors +1 and -1 can be considered as denoting True and False, respectively, in an open world knowledge base.

In the language Prolog (2), the closed world assumption prevails (unless the language is used to simulate a system such as Emycin), and the property that a fact is false if it cannot be deduced is normally provided via a library definition of a "not" symbol. (1, 8) have shown that this leads to reasonable logical behavior under some assumptions which are acceptable in practice. Proof extraction can be used to cause any knowledge base written in Prolog to provide explanations of yes answers (13). However, when such a knowledge base answers no to a question, it does so by failing to find a proof. Hence proof extraction cannot be used directly to explain why something is not so.

This paper describes how a modified interpreter, called Prolog/Exl, assigns a procedural meaning to some programs which lack this in present Prologs, and how explanations of both failure and success are generated.

II DECLARATIVE AND PROCEDURAL MEANINGS

This section assumes some acquaintance with the Prolog language (2). Our notation for a Prolog clause follows (9) in using '<-' and '&' for 'if and 'and', respectively. Variables start with a capital letter, whereas constants do not. (14) describes the declarative and procedural ways of viewing a Prolog program.

The majority of Prolog programs which are declaratively reasonable also behave as expected procedurally. However, under standard interpreters (6, 9), some return no answer. By way of example, consider

```
Example 1
   fly(Jfk, Chi) <-
     fly(Jfk, Bos) & fly(Bos, Chi).
   fly(Jfk,Bos) <- flight(Jfk,Bos).
     flight(jfk, bos).   flight(bos, chi).
     flight(bos, sfo).   flight(chi, sfo).
     flight(sfo, lax).
```

Declaratively, this program states that, in addition to the assertions, fly(jfk, sfo) also holds. Procedurally, trying to show this yields an unbounded recursion on the first clause, so no answer is computed.

With present interpreters, such a program must normally be rewritten e.g. by interchanging the first two clauses, or as

```
Example 2
    fly(Jfk,Bos) <- flight(Jfk,Bos).
    fly(Jfk, Chi). <-
      flight(Jfk, Bos) & fly(Bos, Chi).
      flight(jfk, bos).   flight(bos, chi)
      flight(bos, sfo).   flight(chi, sfo)
      flight(sfo, lax).
```

The change is easy enough, and it fixes one problem. However, suppose we add the assertion flight(sfo, jfk) and ask for all of the possible pairs of endpoints of journeys. In standard Prolog no answer is found at all, because the interpreter keeps trying longer and longer proofs of the same journeys. Prolog/ExI finds the correct answer.

While it is possible to now rewrite this program so that it terminates in standard Prolog, there are other more complicated programs which do not return answers. So it seems better to try to change the Prolog interpreter to deal with the problem (4). It turns out that this change is also needed for explaining 'no' answers, as described below.

As has been pointed out in (6), a Prolog interpreter in Prolog can be written as four clauses, one of which has the form

```
demo(G) <- rule(G<-B) & demo(B).
```

Prolog/ExI keeps a record of the rules which it has used to arrive at its current point in the computation. The clause above is replaced by clauses which test a rule before it is applied. A rule is only used if it has not already been used, or if it is a special case of a rule which has been used. Experimentally, this appears to assign the same procedural meaning to existing programs as does a standard interpreter, but it also assigns the expected meaning to a large number of programs (such as Example 1) which do not normally terminate.

III EXPLAINING YES AND NO ANSWERS

To explain a yes answer, a Prolog interpreter in Prolog can be adapted to accumulate a proof tree during execution, by adding one argument to the 'demo' predicate. The proof tree (or part of it) can then be used as an explanation (14).

Now consider the question 'can one fly from Ihr to lax ?'. Based on Example 2, the answer is no, as there are no flights out of Ihr at all. Unfortunately, an interpreter which gathers explanations in the manner just outlined yields no explanation of this. The computation simply fails, so some other approach is needed.

To explain both yes and no answers, Prolog/Ex1 proceeds as follows. First, it checks to see if the answer is yes, using the approach just outlined. If there is no proof, then, instead of failing, it proceeds to explore possible partial proofs in which certain steps are assumed to succeed (even though in fact they fail). These steps are marked as conditional, and are printed with a question mark in an explanation tree, to indicate where the failures occur. Assuming that a failing goal has succeeded potentially opens up an unbounded computation. Hence the technique described in Section II is used to limit the computation.

In finding explanations, Prolog/ExI follows several guidelines about what kind of explanation is likely to be a help. For a yes answer, it finds a shortest explanation. For a no answer, it finds a conditional explanation in which the first assumption is as deep as possible. Also, for a no answer, if there is a constant in the question, it makes sure that the constant is in the knowledge base and is reachable from the question; otherwise it generalizes the question by changing all other constants to distinct variables.

To see how this works, consider the behavior of Example 2 when interpreted by Prolog/ExI.

If we ask to fly from jfk to lax, we get

```
fly(jfk,lax)
 flight(jfk,bos)
 fly(bos,lax)
  flight(bos,sfo)
  fly(sfo,lax)
   flight(sfo,lax)
```

This can be read as explaining that we can fly from jfk to lax because: there is a flight from jfk to bos and we can fly from bos to lax, and so on. Note that this is the shorter of two possible proofs. The tree can be used to synthesize an explanation in English as in the work of Weiner (15), or can be directly mapped into quasi-English as in the SYLLOG system (12).

If we ask to fly from sfo to Ihr, we get

```
fly(sfo,lhr)
 flight(sfo,lax)
 fly(lax,lhr)
  flight(lax,  )  ?
  fly(  ,lhr)  ?
```

which says that this would be possible, via lax, except that there are no flights out of lax, and there is no sequence of flights into Ihr. If we ask to fly from Ihr to lax, we get

```
fly(lhr,lax)
 flight(lhr,  )  ?
 fly(sfo,lax)
  flight(sfo,lax)
```

indicating that there are no flights out of Ihr. (14) gives the behavior of Prolog/ExI on a knowledge base which is more realistic than Example 2.

The combined techniques of section II and of this section lead to behavior that is reasonably helpful. This behavior is built in to the Prolog/Exl interpreter, so it is available to all knowledge bases without extra effort on the part of the people who write the rules.

IV CONCLUSIONS

When a knowledge base answers a question, its rules are combined by pattern matching, in ways that the writers of the rules may not have foreseen in detail. So, both for the people who provide the rules and for other users, it is important that the knowledge base be able to explain the reasoning that leads to its results, i.e. that it be able to provide some information about what instances of what rules have been used.

If a knowledge base answers yes to a question under a standard interpreter, then an explanation can be produced either by a compiled method (13), or by an interpreter modified to accumulate a proof tree. However, there are declaratively reasonable programs which have no procedural meaning in standard Prolog, that is, certain questions do not yield answers.

If, in standard Prolog, the answer to a question is no, then the answer is the result of a failure to find a proof of a yes answer. All of the rule matches made during the attempted proof are discarded, and no explanatory information remains.

This paper has described Prolog/Ex1, an inference engine which assigns a procedural meaning to many programs which do not produce an answer in standard Prolog. Prolog/Exl provides explanations of both yes and no answers. A 'no' explanation either indicates that some object is missing from the knowledge base, or that while the necessary objects are present, some relationship between them is missing.

Prolog/Exl allows one to write purely declaratively in many cases in which standard Prolog does not, and it provides explanations of both yes and no answers. When there are several explanations for an answer, one which is likely to be useful is chosen.

V ACKNOWLEDGEMENTS

It is a pleasure to acknowledge that this work has benefited from conversations with Chin Chang, Se June Hong, Antonio Porto, David H. D. Warren, and Walter Wilson.

REFERENCES

(1) Clark, K. L. Negation as failure. In Logic and Data Bases , (H. Gallaire and J. Minker, Eds), Plenum Press, 1978, 55-76.

(2) Clocksin, W. F. and C. S. Mellish, Programming in Prolog. Springer-Verlag, 1982.

(3) Codd, E. F. Relational completeness of data base sublanguages. In Data Base Systems , (R. Rustin, Ed), Prentice Hall, 1972, 65-98.

(4) Kowalski, R. A. Logic programming. Report, Department of Computing, Imperial College, London, 1982.

(5) Michie, D. Game playing programs and the conceptual interface. ACM Sigart Newsletter No 80, 1982, 64-70.

(6) Peirera, L. M., F. C. M. Pereira and D. H. D. Warren, User's guide to Decsystem-10 Prolog. Occasional Paper No. 15, Department of Artificial Intelligence, University of Edinburgh, 1978.

(7) Pereira, L., P. Sabatier and E. Oliveira. ORBI - an expert system for environmental resource evaluation through natural language, Proceedings of the First International Logic Programming Conference , Faculte des Sciences de Luminy, Marseilles, France, 1982, 200-209.

(8) Reiter, R. On closed world data bases. In Logic and Data Bases , (H. Gallaire and J. Minker, Eds), Plenum Press, 1978, 55-76.

(9) Roberts, G. M. An implementation of Prolog. M.S. thesis, Department of Computer Science, University of Waterloo, 1977.

(10) van Melle, W. et. al. The Emycin reference manual. Report STAN-CS-81-885, Department of Computer Science, Stanford University, 1981.

(11) Wallis, J. and Shortliffe, E. H. Explanatory power for medical expert systems: studies in the representation of causal relationships for clinical consultations. Report STAN-CS-82-923, Department of Computer Science, Stanford University, 1982.

(12) Walker, A. D. SYLLOG: A knowledge based data management system. Report No. 034, Computer Science Department, New York University, 1981.

(13) Walker, A. D. Automatic generation of explanations of results from knowledge bases. Report RJ3481, IBM Research Laboratory, San Jose, California, 1982.

(14) Walker, A. D. Prolog/Exl, an inference engine which explains both yes and no answers. Report RJ3771, IBM Research Laboratory, San Jose, California, 1983.

(15) Weiner, J. L. BLAH, a system which explains its reasoning. Artificial Intelligence 15, 1980, 19-48.