The pragmatics of Prolog: some comments

by

E.W. Elcock

The University of Western Ontario

London, Canada

N6A 5B9

Abstract

Logic programming and Prolog in particular have done much to illuminate the relationship between logic and computing. The relationship between logical consequence and effective construction is, however, very subtle, and it seems appropriate (even if not entirely novel) to continue to be concerned about too simplistic an approach to the difficulties which face any assertative programming language. This note attempts to focus some of these difficulties for Prolog in the context of a simple but rewarding pedagogic example.

(Keywords: Logic programming; Prolog; programming methodology, pragmatics)

Introduction: "between the expectation and the reality lies the shadow"

The paper comments on the view that Prolog, as an exemplar of logic programming, is a candidate for a specification language and as such provides specifications with a declarative (standard model theoretic) reading, but with the bonus that such specifications can be re-interpreted procedurally and without change as providing implementations of the specifications.

As a specification, a Prolog program [A,G] is to be thought of a sequent A => G. It is well-known, however, that Prolog is an incomplete system: that is, there exist Prolog programs [A,G] where A is a sequence of Horn clauses and G a conjunction of predications such that the corresponding clausal sequent A => G is a true sequent and yet the Prolog program [A,G] does not terminate successfully. A simple example illustrating this incompleteness is the Prolog program [A,G] where A is the sequence of clauses

1. mem(U,[V|L]) :- mem(U,L)

2. mem(U,[U|L])

and G is the goal statement

mem(a,[a|M])

In executing the goal statement Prolog repeatedly uses clause 1 in the procedure for list membership generating the infinite sequence of goal statements

mem(a,[a|M])
mem(a,M)
mem(a,M1) where M is bound to [V1|M1]
mem(a,M2) where M1 is bound to [V2|M2]

etc.

Prolog does not prove the true sequent A => mem(a,[a|M]). In fact, with 'mem' specified with the ordered pair of clauses above in some more general sequence of clauses A , Prolog will not establish the truth of any sequent involving a call of 'mem' on a list with variable tail.

In this example a simple reordering of the clauses of A would result in acceptable computational behaviour. Indeed, the set of Prolog proofs generated with the reordering is a superset of those generated with the original ordering. In this particular example, with A the sequence of clauses

1. mem(U,[U|L])

2. mem(U,[V|L]) :- mem(U,L)

the membership relation is very well behaved and acts synthetically (constructively) as well as analytically. Thus, if G is a goal such as

mem(a,[b|M])

then the Prolog program succeeds with, for example, M bound to [a|M'].

Prolog programmers might rationalize the problem of which this example is a symptom by insisting that, although one wishes to take advantage of the model theoretic semantics of Horn clauses in viewing a Prolog program as a lucid specification, one should be willing in Prolog as in any other language, to rewrite (transform) one's specification, now viewed as a program, with the pragmatics of the procedural interpretation (defined by the particular interpreter or whatever) in mind.

In the particular case of 'mem', and with the procedural interpretation of Prolog firmly in mind, one might rationalize away any unease with some argument that "it's obvious that one should have given the base case of the recursion first" and in this Prolog is no worse than a "conventional" applicative language where the "same thing" might have happened. Certainly Prolog programmers are well aware of the problem and acknowledge it (see for example Clocksin & Mellish, 1982). My readings, however, lead me to believe that many still do not treat the phenomenon with the seriousness it deserves. This note exploits one of my own five-finger exercises using Prolog in the hope of drawing further attention to the problem.

97

Permutations of a problem

Let's now turn to a more focal example: that of using Prolog to write a specification of a solution of the eight queens problem. We specify a board position by an ordered pair of row and column numbers (r,c) 1=<r,c=<8 . We wish to specify the set of subsets of size 8 such that no two members of a subset lie on the same row, column or diagonal. We take advantage of the fact that the interpretation of a sequence of the eight numbers 1 to 8 as a set of ordered pairs (r,c) , where c is the r'th number in the sequence. guarantees that no two members of the set have the same row or column number. With this representation of subsets we simply have to restrict the sequences to be such that no two (r,c) pairs in the represented subset are on the same diagonal.

With this preamble we might begin to specify a solution to the eight queens problem with the Horn clause

```
1.  queens(Q) :- perm([1,2,3,4,5,6,7,8],Q) , dsafe(Q)
```

with the intended interpretation that perm(M,N) specifies that the lists M and N stand in the (symmetrical) relation permutation to one another, and that the predicate 'dsafe' will be suitably specified to capture the intended interpretation discussed above.

So far, so good: the specification has a very clear model theoretic (declarative) semantics contributing to our intended interpretation - still to be filled out by specifications of 'perm' and 'dsafe'. Let's look at the following specification taken from Clark and McCabe's treatment of the eight-queens problem (1979):

```
1.  perm([],[])

2.  perm(L,[U|M]) :- inserted(U,L,L1) , perm(L1,M)

3.  inserted(U,[U|L],L)

4.  inserted(U,[V|L],[V|M]) :- inserted(U,L,M)
```

where 'inserted(U,L,L1)' has the intended interpretation that the list L is the list L1 with the element U

inserted at some (arbitrary) position.

Again these specifications could be claimed to have clear and acceptable declarative readings. We will assume that 'dsafe' can be equally nicely specified - it has no detailed pedagogic role to play in our example.

We seem then to have exploited the model theoretic semantics of Prolog to obtain a very clear and complete specification of a solution to the eight-queens problem by the set of Horn clauses 1.2.... Let us call this set of sentences "A" . The existence of a solution to the eight queens problem could now be asserted by the sequent "A => queens(Q)" .

Prolog would instantiate Q properly. However, if clause 1 of the specification were changed to the apparently equivalent clause

```
1.  queens(Q) :- perm(Q,[1,2,3,4,5,6,7,8]),dsafe(Q)
```

leaving everything else unchanged, then Prolog would <u>not</u> instantiate Q .

This remark is not intended as a criticism of Clark & McCabe, but to draw attention again to the difference between possible expectations and the reality. One has the expectation that a satisfactory axiomatization of 'perm' would necessary capture the symmetry of the relation. The Prolog implementation of the axiomatization by Clark & McCabe does not. In the context of its sole use in a particular axiomatization of the eight-queens problem, the effects of asymmetry have been nullified: in general, however, this might be treating the symptom rather than the disease and would become increasingly opaque in more complex problems involving deeper nestings of axiomatized relations.

The difficulty of the Clark & McCabe axiomatization lies in the fact that if one backtracks to a call of perm(M,L) where M is a variable, then clause 4 for inserted is repeatedly used, each use generating a candidate permutation M consisting of a list with one more uninstantiated variable at its head and an uninstantiated variable tail, each of which candidate permutations finally fails the call perm(M,[]) - as, of course, it should!

It might be thought this problem with 'perm' could be olved by using meta-logical features of Prolog in some pecification such as:

```
1.  perm(L,M) :- nonvar(L) , ! , perm1(L,M)

2.  perm(L,M) :- nonvar(M) , ! , perm1(M,L)

3.  perm1([],[])

4.  perm1(M,[U|L]) :- inserted(U,M,N) , perm1(N,L)

5.  etc., etc.
```

ith the intention that "perm1" is only called with an ppropriately instantiated argument pair such that "inserted" s well-behaved. This specification would certainly "solve" he original problem associated with the eight-queens pecification: however, the new specification of "perm" ehaves in a similar way to the first for pairs of calls such s "perm([1,2,3],[2|L])" and "perm([2|L],[1,2,3])".

In order to emphasize the point made earlier about the otivation of this brief note, a digression is in order. The ollowing is a quotation from comments by an unknown referee presumably chosen for his expertise) of an earlier version f this note:

> "...;for that matter any Prolog programmer knows or should know, that if he wants his predicate to work independently of the data flow he must be careful; hence program ... is not to be written if one knows that L can be a free variable or "infinite" (i.e. end up with a free variable); further the problem is not necessary with "perm", it can be argued that it is with "inserted ": one should then write:
>
> ```
> inserted(U,V,L) :-
>           not var(V), ! , insert(U,V,L)
> ```
>
> where "insert" is given by
>
> ```
> insert(U,[U|L],L)
>
> insert(U,[V|L],[V|M]) :- inserted(U,L,M)
> ```

Admittedly this is a partial solution, but is it correct in all cases where inserted is called; in particular perm([1,2,3],[2|L]) gives the two correct answers for L and M ; similarly for perm([2|L],[1,2,3]) which fails ... Hence there is a natural way to get it right." (My underlining - E.W.E.)

Adopting the referee's suggestion, the specification of "perm" becomes:

```
perm([],[])
perm(L,[U|M]) :- inserted(U,L,L1) , perm(L1,M)
inserted(U,V,L) :- not var(V), !, insert(U,V,L)
insert(U,[U|L],L)
insert(U,[V|L],[V|M) :- inserted(U,L,M)
```

Note that the referee has essentially addressed the subproblem of non-termination by making one of "perm(M,L)" and "perm(L,M)" fail! I did not and still do not regard this as a "natural" way to get "it" right!

Finally, to avoid potential misunderstandings let me stress that this somewhat curious digression has been made to emphasize that my men are not all straw! Some are flesh and blood and refereeing!

Returning to the symmetry problem: in all cases the incompleteness stems from the potential for generating objects from an infinite domain by backtracking. Both the specification of "mem" at the beginning of the paper and our specifications of "perm" give trouble for this reason.

In the case of "mem" the difficulty was removed by a reordering of clauses with the result that no new candidate was generated "unnecessarily". In the case of "perm" the problem is deeper: it cannot be solved by reordering nor by meta-logical wizardy - which indeed addressed the "wrong" problem. Equally important, this kind of incompleteness is potentially difficulty to detect - particularly when the calls to an offending m-ary relation are part of higher relations themselves possibly with completeness constraints of their argument tuples. Thus, in our introductory pedagogic context, the call of

"perm([1,2,3,4,5,6,7,8],Q)" comes from the body of the specification of "queens". Although, as already mentioned, once having diagnosed our difficulty, it is not onerous to change the call to "perm(Q,[1,2,3,4,5,6,7,8])" , one could easily construct more sophisticated examples where the choice of appropriate orderings of argument tuples could become quite a tricky problem.

In the case of our illlustrative example of "perm", and having identified that the difficulty stems from the potential for "inserted" in the specifications above to generate an infinite sequence of objects each of which possesses a property which is going to lead to failure, we can see that it is possible to respecify "perm" to make this impossible. There are two interestingly different ways to do this.

The property in the "perm" specification is that each of the generated lists, L , say, has a length one greater than its predecessor and that the initiation of backtracking takes place by a failure of "perm(M,[])"! Recognition of this motivates the specification:

1. perm(L,M) :- samelength(L,M) , perm1(L,M)

2. samelength([],[])

3. samelength([U|L],[V|M]) :- samelength(L,M)

4. < clauses specifying "perm1" as in Clark & McCabe specification above. say >

With this specification, any call of "perm" with an argument tuple which fixes the common (finite) length of the argument lists will lead to "inserted" being called in a context in which generation of an infinite sequence of objects cannot occur. For example, one of our earlier "problem" calls "perm([2|L],[1,2,3])" would now result in "perm1([2|L],[1.2,3])" being called in an environment in which L is bound to the list [X,Y]. In effect we have a call of "perm1([2,X,Y],[1,2,3])": which call does not permit infinitary generation.

One might, somewhat impudently, present the specification with the motivation that the intended procedural reading is to be "well, we'll do a quick check that the two argument lists are indeed globally consistent with the relation of permutation before we get down to crossing the i's and dotting the t's" !!! Indeed, in the case where L and M are both explicit finite lists then "samelength" acts like this (apart from the tongue in cheek adjective "quick"). More generally however "samelength" acts to construct the most general finite lists L and M which can satisfy "samelength(L,M)" and it has been introduced into the specification for just this reason.

A second (and more "honest"?) way to obtain an axiomatization which is symmetric under the Prolog interpreter is to use a subtler kind of redundancy and write the 'constructive' axioms:

```
perm([],[])

perm(X,Y) :- perm(X1,Y1) , inserted(U,X,X1) ,
             inserted(U,Y,Y1)
```

with the previous axiomatization of "inserted".

Here we simply have the embarrassment that each of the permutations is generated twice!

"This is a long cautionary tale" said the mouse.

The Prolog "perm" saga does not end here. What happens if one wants the set of permutations of some finite list say?

Sets of consequences in Prolog are handled by a non-logical operator "set-of" which essentially explores the whole potential sequent space aggregating appropriate instantions of variables in provable sequents.

Certainly "set-of" works with the axiomatization of "perm" using the covert "samelength" device within the domain of this device.

However, "set-of" does not work with the more "honest" axiomatization immediately above and for the same reason as for previous failures: "set-of" eventually enters an infinite search space, and we once again have the problem of non-termination.

We see here a subtle interaction in Prolog of incompleteness (in the obvious sense), and non-logical operators specifically introduced to do what otherwise couldn't be done!

An adjournment

Having got the bit between one's teeth and with the success of 'samelength' to motivate one, one can return to the original attempt at a direct symmetric specification of "perm" and try

```
perm([],[])
perm([X|L],[X|M]) :- perm(L,M)
perm([X|L],[Y|M]) :- inserted(X,M,M1,Y,L,L1) ,
          perm(L1,M1)
inserted(X,[X|M],M,Y,[Y|L],L)
inserted(X,[X|M],M,Y,[Y1|L1],[Y1|L2]) :-
          inserted(X1,M,M1,Y,L1,L2)
inserted(X,[X1|M1],[X1|M2],Y,]Y|L],L) :-
          inserted(X,M1,M2,Y1,L,L1)
inserted(X,[X1|M1],[X1|M2],Y,[Y1|L1],[Y1|L2]) :-
          inserted(X,M1,M2,Y,L1,L2)
```

where inserted(X,L,L1,Y,M,M1) has the intended interpretation that L(M) is the list L1(M1) with X(Y) inserted in it. This is Prolog - symmetric and works with "set-of". (Of course, one should prove these statements?)

The necessity for the 6-ary function and the multiplicity of cases in the model theoretic reading rather detract from any sense of achievement!

Summary

The late Christopher Strachey told the story that whenever he gave talks on his design for the language CPL, he would inevitably be asked "but can it do so-and-so?" Strachey claimed that as the designer of a good programming language there were

only two possible answers he could give and they were either "of course it can!", or "of course it can't!"

One of the difficulties with Prolog is that it does not meet this criterion and this note has attempted to give a simple example of an important way in which it fails. In a phrase: Prolog holds out promises it cannot fulfill. In particular, to have to consider potentially difficult proofs of termination of procedural readings of obviously true sequents, seems to run counter to one's intuition of what "logic programming" is all about. One of the reasons for the author's concern is that it is like Absys (Foster, 1969) in this. Examples of other ways in which Prolog fails to meet Strachey's criterion could be given. It may well be that there will emerge brands of logic programming languages that will be both pragmatically useful and which will indeed meet Strachey's criterion. It is consistent with the goals of much current research in Artificial Intelligence that the cause of difficulties arising from a first tentative specification might be automatically diagnosed and rectified by a suitable respecification and, as an issue in the study of knowledge representation and use, some form of the problems identified with Prolog above will have to be faced as part of that study as such.

In the absence, however, of substantive progress on such issues it might be better to recognize that the goals of logic and the goals of programming should be regarded as essentially different unless proven otherwise. The goal of logic as usually conceived is to exhibit what things follow from what. The goal of programming as usually conceived is to exhibit how to construct something from other things. Although "what follows from what" certainly provides the framework in which a construction is demonstrated to be valid, to call this validation process "control", at least in the simplistic sense of current computational control structures, and to regard Prolog programming as "logic plus control" e.g. Kowalski, 1979, is to stretch a good catchphrase too far. In what sense, for example, is it appropriate to regard the 'samelength' assertion as a control component of the specification of 'perm' in the example above? Like

most provoking catchphrases, "programming as logic plus control" can be given interesting interpretations. However, for now a better catchphrase, if one wants catchphrases at all, might be that "logic is (Prolog) programming minus control" : a Prolog program, [A,G] , terminating or not, stripped of a particular procedural semantics with its particular concomittant 'control', and re-interpreted as a sequent A => G , is always, if true, demonstrably true in first order logic. (The asymmetry in the two catchphrases is, of course, only in the eye of the believer!).

As mentioned in the introduction, although the declarative aspect of computational text is very important and has been increasing illuminated by the study of the relation between logic and programming, the relationship between consequence and construction is very subtle, and its subtlety must be respected.

The content of this note and, in particular, the relationship between consequence and construction, is being elaborated in a further technical report in preparation. The work is being conducted under Operating Grant Number A9123 from the Natural Sciences and Engineering Research Council of Canada.

References

1. Clark, K.L. & McCabe, F.G. (1979). The control facilities of I.C. Prolog, Expert Systems in the Micro Electronic Age (D. Michie, ed), Edinburgh University Press.

2. Clocksin, W.F. and Mellish, C.S. (1981). Programming in Prolog. Springer-Verlag. Berlin.

3. Foster, J.M. and Elcock, E.W. (1969). Absys 1: an incremental compiler for assertions: an introduction. Machine Intelligence 4, (Eds. Meltzer, B. and Michie, D.). Edinburgh University Press. Edinburgh. pp. 423-429.

4. Kowalski, R.A. (1979). Algorithm = Logic + Control. C.A.C.M. Vol. 22, No. 7. pp. 424-436.