An Introduction to MU-PROLOG

by

Lee Naish

Technical Report 82/2
(Revised July 1983)

Department of Computer Science
University of Melbourne

Abstract

As a logic programming language, PROLOG is deficient in two areas: negation and control facilities. Unsoundly implemented negation affects the correctness of programs and poor control facilities affect the termination and efficiency. These problems are illustrated by examples.

MU-PROLOG is then introduced. It implements negation soundly and has more control facilities. Control information can be added automatically. This can be used to avoid infinite loops and find efficient algorithms from simple logic. MU-PROLOG is closer to the ideal of logic programming.

Keywords and Phrases:

PROLOG, MU-PROLOG, negation, control facilities, coroutines.

CR Categories:

I.2.3 [Artificial Intelligence]: Deduction and Theorem Proving - Logic Programming.

D.3.2 [Programming Languages]: Language Classifications - Very High-Level Languages.

1. Introduction

This paper is intended to give an overview of the MU-PROLOG system in relation to other PROLOG systems. Readers who are interested in the details of the facilities provided should consult the reference manual [Naish 83]. In this paper a variety of aspects of PROLOG are discussed. The comments made are true of most PROLOG systems and specifically are true of DEC-10 PROLOG [Pereira and Warren 79].

The second section of this paper describes two areas where PROLOG falls short of being an ideal logic programming language. They are the implementation of negation, which affects the correctness of programs, and control facilities, which affect efficiency and termination. Section three introduces MU-PROLOG and illustrates how its facilities partially overcome these weak points of PROLOG.

2. Deficiencies of PROLOG

The ideas of logic programming have been summed up in Kowalski's "equation"

Algorithm = Logic + Control

[Kowalski 79]. Ideally, one should be able to specify a problem in simple logic and the logic programming system should find an efficient algorithm to solve it. PROLOG has two problems here. Firstly, negation cannot be expressed using only Horn clauses, on which PROLOG is based. Secondly, the simple left to right evaluation of PROLOG often results in inefficient algorithms.

For these reasons PROLOG needs non-logical primitives, notably cut, which can be used to implement something close to negation and can be used to provide extra control. The non-logical primitives often detract considerably from the simple declarative nature of programs and make verification far more difficult.

2.1. Negation in PROLOG

Initially, we should define what is meant by negation in PROLOG. Horn clauses can only be used to deduce positive information: they tell us what is true but not what is false. One way of dealing with negation in this context is the "closed world assumption" [Reiter 78]. That is, everything that cannot be proved true is assumed to be false. This cannot easily be implemented, so PROLOG uses a weaker rule, "negation as failure" [Clark 78]. A goal is assumed to be false if the interpreter finds a finite proof that the goal is unprovable (that is, the goal fails).

Unfortunately, very few PROLOG systems implement negation as failure soundly. In most cases, not and not equals (\=) behave as if they are defined in PROLOG as follows.

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

X \= Y :- not(X = Y).
```

If we have a goal ?- not(p(X)) then the first clause of not is tried and p(X) is called. If it fails, then the interpreter backtracks and the goal succeeds using the second clause of not. This is what we want since the failure of p(X) means it is false by the negation as failure rule. However, if p(X) succeeds then the cut is called, followed by fail. Because of the cut, the second clause is not tried and so the call to ?- not(p(X)) fails. This may not be correct.

The call to not should, ideally, try to find an X such that p(X) fails and if there is no such X then not(p(X)) should fail. What the call to not actually does is to look for an X such that p(X) succeeds and if there is such an X, it fails. Thus not(p(X)) is implemented as "for all X, ~p(X)" rather than "there exists X such that ~p(X)". It can be proved that variable-free calls do work soundly [Clark 78] (see also [Lloyd 82]), but those with variables can cause problems as the following examples illustrate.

```
?- X \= 2, X = 1.         fails
?- not(X \= 2), X = 1.  binds X to 1 and succeeds.
```

Errors such as these may occur deep inside some computation with disastrous consequences.

To ensure a procedure with not (or \=) as a subgoal works correctly, it is sufficient to demand that when not is called, all the variables in the call are bound. This often restricts the way in which we can call the procedure. For example, the procedure q below should only be called with the first argument ground.

```
q(X, Y) :- not(p(X)), r(Y).
```

The correctness of many PROLOG procedures depends on the way in which they are called. They are not general-purpose, and if they are called in the wrong way, incorrect answers may result.

2.2. Control Facilities of PROLOG

The basic control facilities of PROLOG are just the ordering of clauses and atoms within clauses. Once a program has been written in a particular way, the clauses and sub-goals are always tried in the same order. This restriction can be partly overcome by adding non-logical primitives such as cut and var, but this has undesirable consequences in terms of correctness, clarity etc.

Lack of a clever control component means that simple logic tends to lead to inefficiency. To achieve an efficient algorithm, programs must often over-specify the problem. The result of this extra programmer effort is more complex, less understandable code. Another symptom of poor control is PROLOG's propensity for infinite loops. Procedures can be efficient and terminate for some types of calls but not for other, quite reasonable calls. This again prevents many PROLOG procedures from being general-purpose.

The following simple example illustrates some of these problems. The task is to implement a predicate append3(A, B, C, D) which is true if list D is list C appended to list B appended to list A. The simplest and most obvious way to write it is using the well known predicate for appending two lists together.

```
append3(A, B, D, E) :- append(A, B, C), append(C, D, E).

append([], A, A).
append(A.B, C, A.D) :- append(B, C, D).
```

Now, given the goal

```
?- append3(1.2.[], 3.[], 4.[], X).
```

the interpreter will execute the first call to append, binding the intermediate variable C to 1.2.3.[]. The second call to append will then bind X to 1.2.3.4.[] as desired and if backtracking occurs the calls fail. Thus append3 is ideal for joining three lists together. However, as many introductions to PROLOG point out, a nice feature is that append can be used for splitting lists as well as joining them. Let us see if append3 can do the same thing.

Consider the goal

```
?- append3(X, 3.[], 4.[], 1.2.3.4.[]).
```

The first call to append will match with the first clause, binding X to []. The second append call will fail immediately and cause backtracking. The first call will then be retried and X will be bound to a list of length one. The second call will eventually fail again, the first will be retried again and X will be bound to a list of length two. This time when the second append is called it succeeds and the final binding of X is 1.2.[].

The order of the time taken is the square of the length X rather than the optimal time which is proportional to the length of X. More serious though, from a practical point of view, is what happens on backtracking. The call to the second append fails but the first one, rather than failing, binds X to a list of length three and succeeds. The second call eventually fails again but X is then bound to a list of length four and so on.

Infinite loops such as this quite common in PROLOG. The typical first reaction is to add a cut at the end of the clause for append3. This does not improve the efficiency and worse, it affects the correctness of some types of calls. Another possibility is to change the order of clauses and calls of append. Unfortunately, no ordering works for both joining and splitting lists. A partial solution is to duplicate the logic of append3 and add non-logical primitives like var, nonvar and cut.

The goal

```
        ?- append3(1.W, X, Y, 2.Z).
```

causes an infinite loop for any order of goals and clauses and even the non-logical primitives don't give a simple solution. For many predicates like append3 the simplest definitions in PROLOG do not work correctly, with reasonable efficiency and termination for all types of calls. The non-logical primitives are sometimes useful but usually affect the correctness, so it is often necesssary to completely change the logic. What is really needed is cleverer control which avoids inefficiency and infinite loops.

3. MU-PROLOG

MU-PROLOG extends PROLOG in both the areas mentioned so far. It provides soundly implemented predicates for negation and an improved control component in the form of a more flexible computation rule. The default is the simple left to right execution of PROLOG but some calls may be delayed and resumed later. The order in which sub-goals are solved has no effect on correctness, only efficiency and termination.

3.1. Negation in MU-PROLOG

There are three predicates provided which implement negation: ~, ~= and if-then-else. The first two are sound versions of PROLOG's not and \= and the third is a substitute for a common use of cut.

As we noted earlier, negation as failure works correctly if the negated goal is ground. The ~ predicate uses this fact by waiting until this condition is satisfied. If the call does contain a variable then the call gets delayed and the variable is marked. When the variable is bound (by some later call) ~ is retried. For example:

```
?- ~(X = 1), X = 2.
```

The first call delays and marks X. The second call is then executed, which binds X to 2. This wakes the first call which is now ground so it is executed and the goal correctly succeeds.

Waiting until the call is ground is more restrictive than we would like, but there are difficulties in implementing more general negation correctly. For the particularly common case of inequality it can be done though. MU-PROLOG's ~= only waits until its arguments are sufficiently (not necessarily completely) instantiated. For this reason the correctness proof of ~= must extend the previous theoretical framework. The same is true of the if-then-else predicate. It waits until the condition is ground, but the condition is only executed once, followed by the then or else part.

Negation in MU-PROLOG is slightly restricted but it is implemented soundly. This makes it closer to logic than PROLOG and easier to verify. If a MU-PROLOG program returns some answer, you can be confident that it is correct.

3.2. Control Facilities of MU-PROLOG

MU-PROLOG allows each user-defined predicate to have its own control information. Each predicate then becomes a self-contained module which should behave sensibly for all types of calls. The control has the effect of sometimes delaying a call when it would normally have succeeded. It takes the form of wait declarations which are best explained by an example.

```
?- wait append(1, 1, 0).
?- wait append(0, 0, 1).
append([], A, A).
append(A.B, C, A.D) :- append(B, C, D).
```

The effect of the wait declarations is to slightly restrict the way append can be called. If we call append, and either of the first two arguments are constructed, then the third argument must not be constructed. If this condition is violated then the call delays. A "1" in a wait declaration means that the corresponding argument in a call may be constructed. A "0" means the corresponding argument must not be constructed. Multiple wait declarations provide alternative ways of calling procedures. For example, the call

```
?- append(X, 3.[], 1.2.3.[]).
```

would cause X to be bound to A.B and the call would succeed. Similarly, for the call

```
?- append(1.2.[], 3.[], X).
```

which binds X to 1.D. However, the call

```
?- append(X, 3.[], Y).
```

binds X to [] and Y to 3.[]. Because no wait declarations have ones as the first and third arguments this call delays. The bindings are undone and X and Y are marked in the same way as with ~. The call will be redone when either one is bound.

The only difference between the PROLOG and MU-PROLOG versions of append is the extra control information. This prevents append from being used inefficiently and, in particular, ensures it will never cause an infinite loop. Furthermore, the wait declarations are easy to write and can even be generated automatically. We have developed a program (three pages of PROLOG) which will generate sufficient wait declarations for most procedures. The method used is to look for potential infinite loops and find a minimal set of wait declarations to prevent them.

With the append3 example the programmer can use exactly the same logic as before then have wait declarations added automatically. No matter what order the append clauses and calls are in, the resulting program runs efficiently and terminates for all the goals given before. The user just provides logic and all the control is supplied by the system. With the same ordering the program is as follows.

```
append3(A, B, D, E) :- append(A, B, C), append(C, D, E).

?- wait append(1, 1, 0).
?- wait append(0, 0, 1).
append([], A, A).
append(A.B, C, A.D) :- append(B, C, D).
```

For joining lists it works exactly as the PROLOG version does. For the call

```
?- append3(1.2.[], 3.[], 4.[], X).
```

the first call to append binds C to 1.2.3.[] and the second binds X to 1.2.3.4.[]. No calls are delayed and the wait declarations have no effect. For splitting lists however, the wait declarations play a vital role.

```
?- append3(X, 3.[], 4.[], 1.2.3.4.[]).
```

The first append call needs to bind X to [] and C to 3.[]. This conflicts with the wait declarations so the call delays and the variables are marked. The second append is then called, binding C to 1.B1 and therefore waking the first call. The first clause of append no longer matches so the first append avoids the mistake of choosing it. Instead, the second clause is used, binding X to 1.B2. Then the recursive call, append(B2, 3.[], B1), is tried. This delays, so we continue with the second append call. The effect is for the two append calls to act as coroutines. The time taken is proportional to the length of X.

This efficient algorithm is basically due to the sensible behaviour of the first append call. Because append has its own control information it does not make a rash guess at the length of X. It waits until the second call has provided more information in the form of bindings to C. Because append does not make wrong guesses the algorithm is more efficient and the infinite loop is also avoided. Another illustration of how append avoids infinite loops is the last example we gave with PROLOG.

```
?- append3(1.W, X, Y, 2.Z).
```

The first append call matches the second clause and binds C to 1.D1. The recursive call, append(W, X, D1), delays rather than succeeding, which would cause an infinite loop. The second append call fails, the delayed call wasn't executed so it is not retried and the first call to append has no more clauses to try so the goal fails.

The final example we give is a program to solve the eight queens problem. The logic is due to Clark, who uses it to illustrate how the control facilities of IC-PROLOG can be used to achieve an efficient algorithm from simple logic [Clark 79].

```
queen(X) :- safe(X), perm(1.2.3.4.5.6.7.8.[], X).

safe([]).
safe(A.B) :- notake(A, B, 1), safe(B).

notake(_, [], _).
notake(A, B.C, N) :- nodiag(A, B, N), M is N + 1,
                         notake(A, C, M).

nodiag(A, B, N) :- B > A, W is B - A, W =\= N.
nodiag(A, B, N) :- A > B, W is A - B, W =\= N.

perm([], []).
perm(A.B, C.D) :- delete(C, A.B, E), perm(E, D).

delete(A, A.B, B).
delete(A, B.C, B.D) :- delete(A, C, D).
```

The position of the eight queens is represented as a list of integers. We take the ith number in the list as the column number of the queen in the ith row. The program states that a list X is a solution to the eight queens problem if it represents a safe position (that is, no queen can take another queen), and it is a permutation of the list of numbers from 1 to 8. The notake predicate checks that a given queen cannot take any queen in the following rows. It calls nodiag, which checks if two given queens, N rows apart, are on the same diagonal. The remaining details are left to the reader.

With this logic, most PROLOG systems would, at best, generate complete permutations and then test whether they are safe. A more efficient algorithm is to place the queens one at a time, testing for safeness at each stage. To get this algorithm in PROLOG we would have to completely change the logic. In MU-PROLOG (and IC-PROLOG) all we need do is add some control information. In MU-PROLOG we need the following wait declarations.

```
?- wait safe(0).
?- wait notake(0, 0, 0).
```

The calls to safe and perm act as coroutines. The initial call to safe delays. When perm decides the position of the first queen, safe is woken. Safe initiates calls to notake and safe, both of which delay. Perm continues, and places the next queen. The notake call is woken and it calls nodiag to check if the first queen can take the second queen. If the position is unsafe the call to nodiag fails, and we backtrack. If the position is still safe then nodiag succeeds and notake calls itself, which delays. The delayed call to safe was also woken by perm. It makes two calls, which delay as before.

After perm has generated the positions of the first N queens, there is one call to safe delayed, and N calls to notake delayed (one for each queen). When a new queen is added, these calls are woken. Each of the calls to notake check if the new queen can be taken. The call to safe creates a call to notake, for the new queen, and another call to safe. When all eight queens have been successfully placed, perm binds the end of the list to [] and all the delayed calls succeed.

There are a couple of points that should be made here. Firstly, we have glossed over the details of how perm generates the positions of successive queens. The version of perm given, constructs the list X before it chooses the position of the next queen (by calling delete). This causes the calls to notake to be woken too early, which results in calls being delayed unnecessarily. This overhead can be eliminated if the definition of perm is changed slightly.

The second point is that the order of the calls to safe and perm is crucial. If perm is called first we get back to PROLOG's inefficient algorithm. There is a general rule in MU-PROLOG, that tests should be called before generates. In this case safe tests the list X, and perm generates it. In PROLOG, generates usually must come before tests, if infinite loops and other errors are to be avoided. In MU-PROLOG, calls just delay, and the test and generate can be efficiently coroutined.

4. IC-PROLOG

IC-PROLOG is probably the best known PROLOG system with improved control facilities. A comparison of MU-PROLOG with the growing number of other similar systems will be explored in a future report. IC-PROLOG has soundly implemented negation and improved control facilities [Clark and McCabe 81], though neither are done in the same way as MU-PROLOG. Control information is specified by adding annotations to the program clauses. There are a wide range of annotations, and for some logic, IC-PROLOG can achieve more efficient algorithms than MU-PROLOG. Most annotations are attached to procedure calls, rather than definitions. This allows different control for different calls to the same procedure.

However, there are advantages in having control attached to predicate definitions too. It makes programs more modular. A procedure can be written, and wait declarations added. When that procedure is used, only logic need be considered: the control has already been added. It is also easier to add wait declarations than it is to add annotations to achieve the same effect. Wait declarations are definitely advantageous when it comes to programs like append3. Despite the control facilities available, it is not easy to write a good version of append3 in IC-PROLOG, without changing the logic.

5. Conclusions

PROLOG has poor facilities for negation and control. This leads to reduced reliability, infinite loops and inefficient algorithms. To get efficient algorithms, programs often must over-specify the problems. Procedures are not general-purpose and programs are less logical than they could be.

MU-PROLOG goes some way to solving these problems. By having a more powerful control component, efficient algorithms can be found using simple logic, and infinite loops can be avoided. Each procedure can become a self-contained module which behaves sensibly for all types of calls. Wait declarations can be written easily or generated automatically. The MU-PROLOG programmer can therefore spend less time working on control and can just concentrate on the logic. Thus MU-PROLOG is a step closer to the ideal of logic programming.

6. References

[Clark 78] Clark, K.L., "Negation as Failure", in Gallaire, H. and Minker J. (Eds.), "Logic and Data Bases", Plenum Press, New York, 1978, pp 122-149.

[Clark 79] Clark, K.L., "Predicate Logic as a Computational Formalism", Research Report 79/59, Dept. of Computing, Imperial College.

[Clark and McCabe 81] Clark, K.L. and McCabe, F., "The Control Facilities of IC-PROLOG", in "Expert Systems in the Micro Electronic Age", D. Mitchie (Ed), Edinburgh University Press, pp 122-149.

[Kowalski 79] Kowalski, R., "Algorithm = Logic + Control", CACM, 22, 7 (July 1979), pp 424-436.

[Lloyd 82] Lloyd, J., "Foundations of Logic Programming", TR 82/7, Computer Science Department, University of Melbourne, 1982.

[Naish 83] Naish, L., "MU-PROLOG 3.0 Reference Manual", Computer Science Department, University of Melbourne, 1983.

[Pereira and Warren 79] Pereira, L.M. and Warren, D., "Users Guide to DECsystem-10 Prolog", DAI Occasional Paper 15, Department of Artificial Intelligence, University of Edinburgh, 1979.

[Reiter 78] Reiter, R., "On Closed World Data Bases", in Gallaire, H. and Minker J. (Eds.), "Logic and Data Bases", Plenum Press, New York, 1978, pp 55-76.