PROLOG CONTROL RULES

Lee Naish

Department of Computer Science
University of Melbourne
Parkville 3052
Australia

ABSTRACT

We present an overview of the many control constructs and heuristics used by PROLOG systems with extra control facilities. Two features of computations rules are used to evaluate and classify them. They are detecting failure quickly (where it is unavoidable) and avoiding failures. By examining current systems in this light, we reach conclusions concerning deficiencies in performance, and how they may be overcome. We propose an idealized computation rule which uses a hierarchy of goals and a breadth first component.

1. INTRODUCTION

There are now many PROLOG systems with more control facilities than conventional implementations. The design of these systems has been justified by examples of how programmers can implement efficient algorithms using simple logic. [Naish 85b] went a step further and showed how some control can be generated automatically In this paper, a shortened version of [Naish 84a], we take a much broader view. We examine many proposed and implemented control primitives and heuristics to identify their strengths and weaknesses. We use the term *control rule* for these individual components of complete computation rules Our attention is restricted to control rules for SLD resolution. We hope the discussion and conclusions here will contribute to the design of logic programming systems with better control components in the future.

The main part of this paper introduces some general properties that we should like computation rules to exhibit. The extent to which each control rule contributes to these properties is discussed and used for a simple classification. Finally, an idealized combination of control rules is suggested. First, however, we give some programming examples which will be referred to in the discussion.

2. PROGRAM EXAMPLES

The following selection of programming examples from the literature illustrates the kinds of problems that can be solved efficiently by using a flexible control strategy.

```
perm([ ], [ ]).
perm(X.Y, U.V) :- perm(Y, V), delete(U, X.Y, Z).

delete(A, A.L, L).
delete(X, A.B.L, A.R) :- delete(X, B.L, R).
```

These procedures define the permutation relation on lists. [Block 83] shows how difficult it is to write a definition of *perm* which works with either argument bound using conventional PROLOG. If *perm* is called with the second argument a variable, the execution of *delete* should proceed ahead of *perm* but if the first argument is a variable, *perm* should proceed ahead of *delete*.

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

safe([ ]).
safe(N.L) :- safe(L), nodiag(N, 1, L).

nodiag(_, _, [ ]).
nodiag(B, D, N.L) :- D =\= N-B, D =\= B-N,
                     D1 is D + 1, nodiag(B, D1, L).
```

Using *perm,* we can write a program to solve the eight queens problem. The desirable form of control discussed most is for *perm* and *delete* to generate the list of queen positions one at a time and for *safe* and *nodiag* to test if the new queen position is safe. If the arguments in the initial call to *perm* are swapped, it is more efficient to delay calls to *delete* and =\= until the end, then do the calls to *delete,* resuming the instantiated calls to =\= at each stage.

```
sameleaves(T1, T2) :- leaves(T1, L), leaves(T2, L).

leaves(leaf(X), [X]).
leaves(t(leaf(X), T), X.L) :- leaves(T, L).
leaves(t(t(LL,LR),R), L) :- leaves(t(LL,t(LR,R)), L)
```

This program can be used to check whether two trees have the same list of leaf tags The desired form of control is for the two calls to *leaves* to coroutine. Whenever one further instantiates the list of leaf tags, the other should check if the newly added tag is the next tag in the other tree. Either call can be the generator at each stage. This program can easily be extended to any number of trees.

```
grandparent(G, C) :  parent(P, C), parent(G, P).

ancestor)!*, C) :- parent(P, C).
ancestor)A, I)) :- parent(P, I)), ancestor)A, P).
```

Here we define the *grandparent* and *ancestor* procedures using *parent,* which we assume is defined with a collection of facts. *Grandparent* can be used to find the grandparents or grandchildren of a given person. However, it is most efficient to reverse the calls to *parent* when finding grandchildren.

Ancestor poses some rather difficult optimization problems. For finding the ancestors of someone, *parent* should always be called first. Calling *ancestor* first causes an infinite loop. In fact, infinite loops can always occur if someone is their own ancestor. There are even more difficulties using the program for finding descendants. [Naish 84a] discusses this further,

3. FEATURES OF COMPUTATION RULES

(1) The one obvious overriding property that we wish computation rules to exhibit is to minimize the size of the search tree. Unfortunately, there are very few cases where we can even find heuristics directly related to the size of the tree. Therefore, in the next two paragraphs we introduce heuristics which are reasonably general, but are useful for the design and classification of implementable control rules.

(2) For goals which can finitely fail, computation rules should select atoms which lead to detecting failure quickly. Several heuristics and some theoretical work have contributed to this area.

(3) There is a slightly more subtle rule which applies more to goals which have solutions. Although the success branches of the SLD tree are fixed, the number and length of other branches is not.

The rule, therefore, is to avoid creation of failure branches (and infinite branches) as much as possible.

4. CONTROL RULES

We now discuss many of control rules mentioned in the literature. They are put into three groups, according to the features mentioned above.

4.1. MINIMIZING THE SEARCH TREE SIZE

Unsurprisingly, this section is fairly small, though with more special case analysis, it could probably be expanded in the future.

4.1.1. Select Calls Which FaU

Sub-goals which match with no clauses should clearly be selected immediately. This rule was implemented in METALOG [Dincbas 80], which continually tested whether any atoms had no matching clauses. No method has yet been found for implementing this rule without significant overheads.

4.1.2. Select Deterministic Calls

By deterministic calls, we mean those which match with only one clause. Selecting deterministic calls is optimal for goals with some solution(s). [Naish 85b) shows how control information can increase determinism which can be detected at compile time. We discuss this further in the section on wait declarations.

4.1.3. Database Queries

Given a goal consisting of calls to database procedures (which only contain facts), [Naish 85b] gives a formula for the number of calls needed to find all solutions. It is a heuristic, based on some assumptions about probabilities of various matches being independent, etc. This formula can be generalized to take account of the number of unifications performed, which depends on the form of indexing used [Naish 85a] It can be minimized to find the best computation rule Calls to large database procedures should generally be delayed until less expensive calls have been done. This generalizes the methods of [Warren 81] and [Stabler 83] and produces the best form of control for *grandparent*.

4.2. DETECTING FAILURE

4.2.1. Call Tests as Soon as Possible

Tests fail more often than other calls. Thus, to detect failure quickly, they should be called as soon as possible. Programmers generally have a good idea of what calls are tests and [Naish 85b) and [Naish 85a] suggest ways of recognising tests automatically. The proposed definition is that a *test* is a) deterministic and does not construct any variables when it is sufficiently instantiated and b) has an infinite number of solutions otherwise. One problem is that if tests are called too soon, they usually create failure branches. This is normally solved by delaying the call if certain variables are uninstantiated When they become bound, the test should be resumed quickly.

4.2.2. Eager Consumers

IC-PROLOG's *eager consumer* annotations [Clark 79] can be used to call tests quickly without creating extra failure branches. Placing an eager consumer annotation on some variable in a sub-goal prevents that sub-goal constructing the variable. The whole computation of the subgoal is delayed if an attempt is made to further instantiate the annotated variable. This has the unfortunate consequence of delaying instantiated tests in cases where the annotated sub-goal calls several tests. For example, if *safe* is made an eager consumer in the eight queens program, only one call to *nodiag* is called when a new queen is added. A similar problem is caused by the restriction that only one sub-goal can be a designated consumer of a particular variable. One advantage of *eager consumers* is the "inheritance" of the annotation to sub-terms. This is useful for the *sameleaves* program.

4.2.3. Fairness

[Lassez 84] shows that SLD resolution is complete with respect to finite failure, assuming a fairness condition. Depth first rules and rules for most primitives which delay calls are unfair. There are two aspects of fairness which could affect practical systems. The first concerns avoiding infinite loops and detecting failure where possible. A fair computation rule could be used when no better heuristics can be found. The second aspect concerns completeness. Several control primitives can delay calls indefinitely, causing incompleteness. With a fair computation rule, all calls would be done eventually.

4.2.4. Breadth First

The simplest way to ensure fairness is to use a *breadth first* computation rule Usually, generators and tests produce and consume (respectively) data structures at similar rates. Typically, one level of recursion corresponds to one level of functor nesting. This implies that a breadth first rule would have a fairly small delay between generating and testing, so failures are found relatively quickly Unfortunately, a strict breadth first rule is very poor at avoiding failure, especially when tests are called before generators.

4.2.5. Pseudo Parallelism

IC-PROLOG's // connective has a declarative reading of "and", but the two (or more) sub-goals it connects are computed in pseudo-parallel. The computation rule alternates between selecting atoms from each of the different sub-computations. The same control has also been used as an example of the power of the meta-interpreter approach to control used by Two-Level PROLOG [Porto 84], If // is used for all and-connectives, the result is a fair computation rule. However, if one sub-computation is a generator and the other contains several tests, the execution of the tests tends to lag behind the generator.

4.2.0. Avoid Left Recursion

This is a goal ordering heuristic, suggested for MU-PROLOG in [Naish 85b]. Actually, left recursion is desirable in some situations, such as *perm* in our alternative eight queens example. The problem is that left recursion is a pathological case for failure detection with a depth first rule, which most current systems use. With a breadth first control rule, failure detection is improved and left recursion is not a problem.

4.3. AVOIDING FAILURE

4.3.1. Freese

The main reason for delaying sub-goals in PROLOG is to avoid creating failure branches and there are very many primitives which enable this. The simplest is *geler* (freeze) of PROLOG II [Colmerauer 82]. *Freeze* is used to delay a sub-goal until a particular variable is bound to a non-variable Because it only delays a single call, the eight queens can be made more efficient than with eager consumers, though freeze is needed for four different sub-goals. However, because the control is not inherited to sub-terms of the variable, the same leaves program cannot easily be made efficient. Also, because freeze only waits for one variable, it is less useful for multi-use procedures and cannot make perm work in both ways.

4.3.2. Lasy Producers

IC-PROLOG's *lazy producers* provide a powerful method of avoiding failure and, to a lesser extent, detecting failure. A lazy producer annotation on a variable in a sub-goal prevents all other calls from constructing the variable. When another call attempts to construct the annotated variable, that call is delayed. The producer is then executed until it binds the variable, then the delayed call is resumed. The choice of which call is resumed does not help avoid

failure but, if the call is a test, the choice helps detect failure. This overlaps with the control provided by *eager consumers* and means that coroutining between a generator and multiple tests is still difficult to implement

4.3.3. Wait Declaration!

Under this heading, we include the wait declarations of MU-PROLOG [Naish 84b) and the algorithm used for generating them automatically [Naish 85b]. We believe it is a major contribution to avoiding failure. The effect of wait declarations is local, like freeze, but they can be used to delay a call until one of several argument sets is sufficiently instantiated. This added flexibility makes it possible for procedures such as *perm* to work in multiple ways. The heuristic also produces the best form of control in goals like the following. The failure producing subgoals *(safe, nodiag* and *perm)* are delayed by automatically generated wait declarations whereas *delete* is not.

? safe(L), nodiag(N, 1, L), perm(Z, L), delete(N, [1,2,3,4,5,6,7,8], Z).

Automatically generated wait declarations also interact very favourably with the rule for selecting deterministic calls first. With the eight queens program, calls to all procedures except *delete* are forced to be deterministic and this can easily be detected by a pre-processor. Using this information, our alternative eight queens control can be automated. However, there are situations where generated wait declarations delay calls unnecessarily or where wait declaration cannot be generated at all (such as *ancestor).* Both these problems can be overcome by fairness The calls should just be given a very low priority, rather than being delayed indefinitely or not handled at all. With this control, *parent* would always be called before *ancestor.*

4.3.4. Delaying System Predicates

In IC-PROLOG, partially instantiated calls to system predicates such as < act as generators, often creating failure branches. In MU-PROLOG, they delay instead, allowing our alternative eight queens control. For completeness, it would be preferable for the system tests to be called eventually, if possible.

5. DISCUSSION

With most systems, the methods available for avoiding failure are not flexible enough. To delay the calls which create failure branches, other calls must be delayed also. This is manifest is two ways. Firstly, IC-PROLOG delays whole sub-computations. Secondly, most primitives only allow sub-goals to wait for a single variable to be bound, even though many procedures can work efficiently consuming several different subsets of their arguments. Wait declarations are an exception. They only delay single calls and are flexible enough to enable multi-use procedures Partly because of this, they can also be generated automatically The deficiencies in the algorithm can be partially compensated for by having a fair computation rule, so calls delayed by wait declarations are still done eventually.

There are also other deficiencies with failure detection, despite this being well understood. Because of delaying whole sub-computations and the single eager consumer limitation in IC-PROLOG, failure detection is impaired when multiple tests are needed. With other systems especially, multiple (potential) generators, such as the same leaves program, are not handled well. Left recursion also causes problems. Both these areas can be improved by using a breadth first rule. This performs slightly worse than a more controlled coroutine approach but requires DO programmer intervention.

Our idealized system has three major features. Firstly, calls which are likely to create extra failure branches are delayed. Secondly, other calls which are likely to fail are called first. Thirdly, the computation rule is fair, so even calls likely to create failure branches are called eventually. We propose a hierarchy of calls as follows:

(1) Tests.
(2) Other deterministic calls.
(3) Nondeterministic calls.
(4) Calls to database procedures.
(5) Calls to procedures for which wait declarations cannot be generated.
(6) Calls delayed by wait declarations.
(7) Delayed calls to system predicates.

The optimal order in which to call the database procedures can be determined and other types of calls should be done in a breadth first manner, for failure finding and fairness. Furthermore, it is desirable that a lower priority call be done after some number of calls (say 1000) of the next higher priority, to ensure fairness.

0. CONCLUSIONS

Current PROLOG systems with extra control facilities have been designed in a fairly ad hoc manner, relying mostly on a few example programs. We have introduced some more general principles on which control rules can be judged. This shows the weaknesses and strengths of current control rules more clearly and should be of use in designing future systems which further exploit the advantages of flexible control strategies.

7. REFERENCES

[Clark 79]
K. L. Clark and F. McCabe, The Control Facilities of IC-Prolog, in *Expert Systems in the Microelectronic Age,* D. Michie, (ed.), University of Edinburgh, Scotland, 1979, 153-167.

[Colmerauer 82]
A. Colmerauer, Prolog-II Manuel de Reference et Modele Theorique, Groupe Intelligence Artificelle, Univerisite d'Aix-Marseille II, 1982.

[Dincbas 80]
M. Dincbas, The METALOG Problem-Solving System. An Informal Presentation, in *Workshop on Logic Programming,* S. A. Tarnlund, (ed.), Debrecen, Hungary, July 1980, 80-91.

[Elcock 83]
E. W Elcock, The Pragmatics of Prolog: Some Comments, *Proceedings of Workshop on Logic Programming,* Algarve, Portugal, 1983.

[Lassez 84]
J. L. Lassez and M J. Maher, Closures and Fairness in the Semantics of Programming Logic, *Theoretical Computer Science 29,* (1984), 167-184.

[Naish 84a]
L. Naish, Prolog Control Rules, Technical Report 84/13, Department of Computer Science, University of Melbourne, 1984.

[Naish 84b)
L. Naish, MU-Prolog 3.1db Reference Manual, Internal Memorandum, Department of Computer Science, University of Melbourne, 1984.

[Naish 85a]
L Naish, Negation and Control in PROLOG, Ph.D. Thesis (in preparation), Department of Computer Science, University of Melbourne, 1985.

(Naish 85b]
L. Naish, Automating Control for Logic Programs, *The Journal of Logic Programming (To appear),* 1985.

[Porto 84]
A. Porto, Two-Level Prolog, *International Conference On Fifth Generation Computer Systems,* November 1984.

[Stabler 83]
E. Stabler and E. W. Elcock, Knowledge Representation in an Efficient Deductive Inference System, *Proceedings of Workshop on Logic Programming,* Algarve, Portugal. 1983.

[Warren 81]
D. H. D. Warren, Efficient Processing of Interactive Relational Database Queries Expressed in Logic, *Proceedings Seventh International Conference on Very Large Data Bases,* Cannes, France, 1981, 272-281.