Computing with Sequences

C.D.S.Moss. Feb 1933

Abstract

All Prolog implementations deal implicitly with sequences of solutions to problems by means of backtracking: if one solution to a subproblem is rejected another is presented. A number of implementations also provide predicates which provide sets or bags (sets with possibly repeated elements) of solutions. But in these case the system finds all the solutions before proceeding. It is suggested here that an implementation of bags, or sequences, which finds only one solution at a time can be integrated easily with existing implementation techniques for Prolog.

This technique has a number of advantages over similar proposals. If it is combined with subprograms which exhibit tail recursion, then one can write logically correct programs to process sets of solutions which do not use extra memory. These can include programs which reduce solutions (e.g. count, sum or average solutions) without using "impure" techniques. However the technique involves little overhead in programs which do not use it, unlike certain other proposals for coroutining.

This will have particular advantage in database applications where large amounts of information must be retrieved serially from secondary storage. It also has potential application for parallelism since the next solution of the subproblem may be pre-evaluated on a parallel processor, without changing the normal interpretation of Prolog clauses.

Introduction

The normal mode of evaluation in Prolog may be termed a "semi-lazy" evaluation of all the solutions of a goal. In other words, one solution is computed to a subproblem, and the computation is then suspended until another solution is required. This produces the very attractive "stack" discipline which characterises Prolog and contributes significantly to its speed and memory efficiency.

But in many cases one wishes to talk about "all" solutions to a problem. A facility is provided in several Prolog implementations (e.g. Warren [1982]) by an evaluable predicate which produces tne solutions as a list:

The predicate "setof(A,B,C)" means that C is a list of the variables A which solve the problem B.

In many situations, particularly if one wishes to nest calls to such a predicate, it is desirable to provide the solutions as a set, in which any duplicate solutions have been removed. But this clearly entails extra work, and presents questions if some of the solutions contain uninstantiated variables (does one want the most general or most specific answers?). Hence some implementations also provide a "bagof" predicate which is defined in the same way, but can contain duplicate answers. But this is also implemented by producing all solutions to the problem at one time and therefore involves allocating (implicitly) enough space to hold all the solutions.

Because of this, many programmers will persist in using the impure "hacks" that were common in Prolog before these predicates were introduced. These involve making temporary assertions in the database to hold information which is not lost on backtracking. Apart from the loss of speed from using these techniques, programs can become obscure, as the technique effectively introduces global variables into a language which avoids their use otherwise. In the context of parallelism, such usage is doubly suspect.

Using Sequences

The introduction of sequences has two parts: an evaluable predicate and a "lazy" way of processing (only) lists. We will introduce a new predicate called "seqof". It has the same definition as "setof" or "bagof" except that solutions produced in the list may be repeated, and the order of solutions is defined by the program. i.e.

seqof(A,B,C) means that the list of all solutions for A in goal B is a list C.

Let us demonstrate the use of this by showing a procedure which computes the average population of all countries in a database.

```
averagepop(A) <- seqof(B, (population(C,D), country(C)), E),
        average(E,0,0,A).

average(nil,0,0,0).
average(nil,A,B,A/B).
average(A.B,C,D,E) <- average(B, C+A, D+1, E).
```

Here "averagepop" computes the average population and "average" is a general procedure for computing averages, working in a "bottom-up" fashion so that tail recursion is applicable. Note that infix function calls to arithmetic predicates are assumed. A definition of all predicates used may be found at the end.

The meaning of this program may be appreciated quite separately from the method of implementation: "seqof" generates a list containing the populations of every country in the database, and "average" acts recursively on this list to produce the average.

However the implementation suggested is as follows: seqof(A,B,C) evaluates B until a single solution is found, when it binds C to A.x (where x is defined below), or if no solution is found then C is bound to nil. Control then passes to the subgoals following seqof - in this case "average" (unless C is already bound fully to a list in which case the next solution of B is found). Execution proceeds normally until an attempt is made to bind a non-variable to the lazy list object, the value called "x" above. If this is normal list-processing, then it will be an attempt to bind it to some term "D.E", or "nil". At this point, control is returned to "seqof" and another solution is attempted. If it is found, then the value D will be instantiated and processing returns to the list consuming procedure.

One valuable aspect of this control mechanism is that the "seqof" procedure can be programmed so that when control returns to it, it can detect whether any valid references still exist to the first solution (because of backtracking points etc), and if not, can delete that solution from the stack. In this way, the global stack space used can be reclaimed and a list of all solutions used without consuming an equivalent amount of stack space.

A Database Example

As a further example of the use of this approach, consider another database query. Buneman et al [1982] consider queries such as:

"find the names of employees who are under 30 years of age and are paid more than the average salary for all all employees."

In a typical database query language this might be expressed:

```
retrieve NAME from EMPLOYEE where AGE<30 and
   SALARY > average(retrieve SALARY from EMPLOYEE).
```

They demonstrate that this can be expressed in a purely functional query language by an expression

```
!EMPLOYEE o |([[AGE,30]o LT, [SAL,AVESAL] o GT] o AND) o *NAME;
```

where "o" represents function composition, AVESAL is a function which computes the average salary, "!" generates a stream of all employees, "|(" restricts this stream by the following predicate, and "*" converts a stream of values into a

character stream.

A system such as Chat-80 (Warren, Pereira 1979) might represent this as a Prolog query in the form:

```
ans(ANS) <= seqof(A, employee(B) & age(B,C) & C<30
                  & salary(B,D) & D>E &name(B,A), ANS)
          & seqof(F, employee(G) & salary(G,F), H)
          & average(H,0,0,E).
```

While this (relational) form of the enquiry is not as concise as the functional form it has an important advantage. The variable "E" which represents the average salary is independent of the expression in which it appears and it is clearly not necessary to compute it for each employee. In the Chat-80 system the query would be rearranged and constraints inserted to ensure that it is only calculated once (Warren 1981). In a functional system, the optimiser has to recognize a "constant subexpression".

The processing of the query clearly involves two passes over the employee relation, which may be presumed to be large and stored in a database. When computing the average salary there is only a need to retain the partially computed average. The second pass generates the names of employees, which will presumably be printed out and also not stored.

Buneman et al point out that there is a further possible optimisation for this query: if there are no employees less than 30 it is unnecessary to compute the average salary. The implementation of sequences proposed here would not easily allow for this.

Implementation Details

This discussion has ignored the vital implementation details of how the stack is organized to handle several different "control points" simultaneously (and one must remember that seqof may be called at several points in direct or indirect recursion). There are in fact a number of possible implementations and which is used may depend on several factors.

1. One can implement full "spaghetti stacks" in which several computations can interleave. This is the approach of Clark and McCabe [1979]. This does not unfortunately mesh very well with existing implementation techniques.

2. One can start a new stack for the seqof predicate separate from the old stack. In fact it is possible to consider this as a completely separate "process", as will be explored below, and in a virtual memory environment where address space is no problem (though real memory is) this may well be the best solution.

3. One can allocate (by some heuristic) a certain space on the stack for the seqof process in addition to that consumed by the first solution and start the following processes above this. On return to the interrupted process, it performs a "context switch" which resets the top of available stack space to the predefined space. If this space is filled, it is then necessary to "stack-shift" the rest of the stack to make space. Fortunately this is an operation which is already done by several implementations to allow for the several different storage areas used by Prolog.

The advantage of methods 2 and 3 (and there may be better methods) is that they do not incur a large time penalty on normal execution. The convenient and efficient stack handling of Prolog is preserved and the normal sequencing of goals is only interrupted when treating a "lazy" list.

It is however necessary to modify the unification routine to recognize a new object - the lazy list. However this does not affect the binding variables to the object, only the attempt to match it with another list object (list functor or nil). Hence the modification is limited to the case of binding a functor to a functor - and even in compiled code this is normally performed by a subroutine. It is to be hoped that with the gradual introduction of microcoding for unification this will not be a significant problem and be outweighed by the saving in space achieved.

Some other uses of this technique

This technique opens up new possibilities for Prolog evaluation, of which three will be explored briefly.

1. Input-Output in Prolog programs is generally handled in a manner which is both semantically impure and obscure from a programming point of view. Consider the following programming fragment found in several systems:

```
get(X) :- repeat, get0(X), X = ' ', !.

repeat.

repeat :- repeat.
```

Here the predicate "get0" unifies the next character in the input (of some unspecified file) with the parameter X. If it is not a space, then the system backtracks to get0, which is not implemented as a procedure with backtrack points. However backtracking to "repeat" always succeeds and the next character of the input is read. Thus "get" reads the next non-blank character from the input. However, both "get0" and "get" are predicates which have side-effects. Backtracking over the input file does not "rewind" the file as one might expect.

Let us suppose that input-output predicates were defined in terms of the seqof predicate. Thus, for instance, a call to the predicate file(A,B) binds B to a character in file A. A call to seqof(B,file(A,B),C) binds C to the list of characters in file A. If this is the predicate used then correct backtracking behavior will be acheived automatically with more economical use of storage. Of course a more sophisticated implementation might yield even greater economy.

2. Consider the following use of seqof:

seqof(A, proc1(B,A), C), seqof(D, proc2(C,D), B).

Here proc1 and proc2 are two processes which each "consume" a list which is their first parameter and produce answers which are their second parameter. If the first parameter is considered a list of input messages, then the behavior of these two processes may considered to be one of message passing from one to the other. Initially proc1 is invoked and produces a message A. Then proc2 is invoked and produces an answer D. When it tries to consume the second message, control is passed back to proc1. Notice that there does not need to be any one-to-one correspondence between the number of messages provided by proc1 and proc2. Also deadlock is easily detected.

3. It is possible to replace any subgoal in a Prolog program by a pair of goals which are equivalent:

goal(A,..,Z) to

seqof(A:..:Z, goal(A,..,Z), C), soln(C,A:..:Z)

where A:..:Z represents a tuple of the variables appearing in the goal and soln is defined conceptually by

soln(A.B, A).

soln(A.B, C) :- soln(B, C).

Thus we may consider any subgoal in a Prolog program as a separate subprocess which generates a sequence of solutions which is then passed on to its neighbors. It is possible to implement these in the most convenient manner: one possibility that is attractive is for a subgoal to produce exactly one more solution than has yet been demanded by the other goals. Then when the next subgoal backtracks there will be a solution immediately available. However the demand for new processes will not grow exponentially as could be the case if indefinite parallelism were allowed.

Conclusion

The introduction of the seqof predicate and the "partially evaluated list" technique has introduced into Prolog a very limited form of coroutining. It has the advantage of preserving

the behavior of programs written for a left-to-right depth first evaluator and avoiding the overhead which has accompanied many other proposals for coroutining. It has the disadvantage of being somewhat "delicate" in its space-saving value: if the user leaves other references to the list at some point after the seqof call, then it may not be possible to discard the intermediate solutions.

Definition of predicates used

In the following definitions, the name of the predicate is given first followed by the number of its parameters; then an English definition of the predicate in which variables A,B,C etc. are used to represent the 1st, 2nd, 3rd etc. parameters of the predicate respectively.

bagof/3 -- the bag of all solutions for A in goal B is a list C.

seqof/3 -- the sequence of all solutions for A in goal B is a list C.

setof/3 -- the set of all solutions for A in goal B is a list C.

averagepop/1 -- the average population of all countries is A.

average/4 -- the average value of sublist A of a list, having a partial total of B from C items in the head of the list, is D.

population/2 -- the population of A is B.

country/1 -- A is a country.

soln/2 -- B is a member of the sequence A.

employee/1 -- A is the identification of an employee

age/2 --y the age of employee A is B

salary/2 -- the salary of employee A is B

name/2 -- the name of employee A is B

References

P. Buneman, R.E. Frankel, R. Nikhil [1982]: An Implementation Technique for Database Query Languages. ACM Trans. on Database Systems. 7/2, pp164-186.

K.L. Clark, F. McCabe [1979]: The Control Facilities of

IC-Prolog. In D. Michie (ed): Expert Systems in the Microelectronic age. E.U.P.

D.H.D. Warren, F.C.N. Pereira [1978]: An efficient easily adaptable system for interpreting natural language queries. D.A.I. Paper 155, Univ. of Edinburgh.

D.H.D. Warren [1981]: Efficient Processing of Interactive Relational Database queries expressed in logic. VLDB Conf. pp272-281.

D.H.D. Warren [1982]: Higher order extensions to Prolog: are they needed? Machine Intelligence 10, pp441-454.