Information Processing Letters 27 (1988) 249–252
North-Holland

28 April 1988

A NEW COMPUTATION RULE FOR PROLOG

Ashok KUMAR and V.M. MALHOTRA

Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur 208 016, India

Communicated by David Gries
Received 13 July 1987
Revised 8 October 1987

Keywords: PROLOG, SLD-resolution, interpreter

Introduction

A common programming paradigm in PROLOG is generate-and-test: a 'solution' is generated and then 'tested'. A solution that fails to satisfy the test is discarded—at least partially—and a new solution generated. Intelligent backtracking [1] has been suggested for reducing the number of calls that a PROLOG interpreter makes during a program execution. Two main drawbacks of an intelligent backtrack are: (i) it does not prevent the first unsuccessful execution of a path, and (ii) it needs to carry too much information to make a good guess.

In this paper we introduce an alternative method for reducing the number of calls. The method achieves an early rejection of a bad solution by scheduling the tests closer to the generators.

1. PROLOG

We assume the reader to be familiar with the common PROLOG terminology [2,5]. Briefly, a PROLOG program consists of a finite set of clauses of the form $A \leftarrow B_1 B_2 \ldots B_n$ $(n \geqslant 0)$ and a goal clause G of the form $\leftarrow B_1 B_2 \ldots B_n$ $(n \geqslant 0)$. Each B_i $(i = 1, 2, \ldots, n)$ is called a subgoal. As an example, we give a simple Sort program below. The program is a typical example of a generate-and-test paradigm in PROLOG.

```
S1:  Sort(X, Y) ← Perm(X, Y)Ok(Y);
P1:  Perm([ ], [ ]);
P2:  Perm([X|Y], [H|T])
         ← Apnd(U, [H|V], [X|Y])
           Apnd(U, V, W)Perm(W, T);
A1:  Apnd([ ], X, X),
A2:  Apnd([X|Y], Z, [X|W])
         ← Apnd(Y, Z, W);
K1:  Ok([X, Y|Z]) ← Ok([Y|Z])Less(X, Y);
K2:  Ok([X]);
         ← Sort([3, 2, 1, 4, 5], Y);
```

Let G be $\leftarrow A_1 \ldots A_m \ldots A_k$ and let C be $A \leftarrow B_1 \ldots B_q$. G and C are said to derive a new goal Q using mgu Θ if:
- $A_m \Theta = A\Theta$ and
- Q is the goal

$$\leftarrow (A_1 \ldots A_{m-1} B_1 \ldots B_q A_{m+1} \ldots A_k)\Theta.$$

The mechanism for selecting a subgoal S of goal G for deriving the next goal Q is called a computation rule. The standard computation rule for PROLOG selects the leftmost subgoal of the goal for the derivation.

PROLOG searches the clauses for the selected subgoal in order of their occurrence in the program. Each time a sequence of derivation fails (i.e., no further derivation is possible), the last derivation in the sequence is undone and the next untried clause for selected subgoal is used to search an alternative derivation sequence. The search ends when a sequence of derivations leading to an empty goal is obtained.

2. Motivation for a new computation rule

The execution trace (sequence of goals) of the Sort program indicates that the test Ok(Y) is called several times—Perm continues to generate new permutations of input [3, 2, 1, 4, 5] and calls Ok(Y) till the search succeeds. Thus, Ok(Y) is called with Y bound to [3, 2, 1, 4, 5], [3, 2, 1, 5, 4],.... Specifically, all permutations beginning with 3, 2 fail to satisfy test Ok(Y). The aim, therefore, is to devise a computation rule that calls test Ok(Y) as soon as the first two elements of the permutation have been chosen.

An interpreter for sequential PROLOG is a partial function. For example, the Sort program will not terminate if the two subgoals in the body of clause S1 are reversed, i.e., Sort(X, Y) $\leftarrow$ Ok(Y) Perm(X, Y). As a result, the arrangement of the clauses and the subgoals in the clause body is, in general, significant and cannot be altered without jeopardizing the execution of the program. In particular, the proposed computation rule for PROLOG must ensure that it produces a result whenever the standard computation rule for PROLOG does. For this purpose, we shall distinguish the leftmost occurrences of the variables in a goal from the other occurrences of the variables in it. For example, in the goal $\leftarrow$ Perm([3, 2, 1, 4, 5], Y)Ok(Y), Perm([3, 2, 1, 4, 5], Y) must be selected before Ok(Y) since it contains the leftmost occurrence of variable Y.

3. A new computation rule

As explained earlier, a computation rule that prefers a 'test' over 'generation' is expected to reduce the demand for computation during interpretation of a PROLOG program. We characterize a 'test' as a derivation of a new goal that does not introduce any binding to the variables in the goal, i.e., the mgu contains only substitutions for variables in the head of the called clause. On the other hand, 'generation' is a derivation causing bindings to some of the variables in the goal.

The proposed computation rule selects a 'test' subgoal, if one exists. If no 'test' subgoal exists, a 'generator' subgoal is selected. If the goal contains neither a 'test' nor a 'generator' subgoal, the leftmost subgoal is selected. The first untried clause for the selected subgoal is used to derive the new goal.

In the remainder of this section we formally characterize tests and generators.

Assignment

Let S be a subgoal and let H be the head of the clause unified with S. Let Θ be a most general unifier (mgu) for S and H. We say that a variable V in S has undergone an assignment if V is bound to a variable in S or to a (nonvariable) term by the mgu Θ. The binding of V to a variable in H is, however, not an assignment.

For example, if $S =$ Apnd([ ], [2, 1, 4, 5], W) and $H =$ Apnd([ ], X, X), then W is assigned the value [2, 1, 4, 5] by the mgu. We represent this as $W :=$ [2, 1, 4, 5].

Leftmost variable and subgoal

A subgoal S is the leftmost subgoal for variable V if S is the leftmost subgoal containing V in the body of the clause that introduced V. All occurrences of V in its leftmost subgoal are leftmost if V does not occur in the head of the clause. The occurrences of V are potentially leftmost if V occurs in the head. A potentially leftmost occurrence of a variable V becomes leftmost if all occurrences of the variable in the head of the clause are in the terms that unify with the leftmost variables of the selected subgoal.

For example, in the clause

$$\text{Perm}([X \mid Y], [H \mid T]) \leftarrow \text{Apnd}(U, [H \mid V], [X \mid Y]) \; \text{Apnd}(U, V, W)\text{Perm}(W, T),$$

the occurrences of variables X, Y, and H are potentially leftmost and those of U and V leftmost in $\mathsf{Apnd}(U, [H \mid V], [X \mid Y])$. W is leftmost in $\mathsf{Apnd}(U, V, W)$ and T is potentially leftmost in $\mathsf{Perm}(W, T)$.

Waiting subgoal

Let V be a nonleftmost variable in subgoal S. Let the next call for S be to a clause with head H. If unification of S with H assigns c to V, then S is said to be waiting on assignment $V := c$. A subgoal may wait on several assignments at a time.

Let a derivation make an assignment $V := t$. The assignment satisfies the wait of S on $V := c$. The set of new waits for S introduced by the assignment is given by $\mathsf{wait}(c, t)$, where

$$\mathsf{wait}(c, t) = \begin{cases} \{\,\} & \text{if } (c \text{ or } t \text{ is a leftmost variable in } S) \\ & \quad \text{or } (c = t) \\ & \quad \text{or } (c \text{ is a variable in } H), \\ \{c := t\} & \text{if } (c \text{ is a variable in } S), \\ \{t := c\} & \text{if } (t \text{ is a variable in } S), \\ \bigcup_{c', t'} \mathsf{wait}(c', t') & \text{if } (c.\text{functor} = t.\text{functor}) \text{ and } \\ & \quad c', t' \text{ are corresponding pairs of arguments}, \\ \textbf{fail} & \text{otherwise}. \end{cases}$$

Table 1
A comparison of the number of calls made during execution by PROLOG programs using the proposed and conventional computation rules; the goals marked by an asterisk (*) do not cause any backtracking

Sequence number	Initial goal	Number of calls	
		Proposed	Conventional
1	Queens(4, X)	108	281
2	Sort([5, 4, 3, 2, 1], X)	329	1749
3	Sort([2, 3, 4, 5, 1], X)	240	1382
4	Sort([5, 4, 1, 2, 3], X)	130	935
5	Sort([1, 2, 3, 4, 5], X)	26	26
6	* Quicksort([5, 4, 3, 2, 1], X)	45	45
7	* Insort([5, 4, 1, 2, 3], X)	29	29
8	* Reverse1(30 elements)	496	496
9	* Reverse2(40 elements)	42	42
10	Node_color(4 nodes)	7	9

Mature subgoal, test, and generator

A subgoal not waiting on any assignment is a mature subgoal. A mature subgoal S is a test if the unification of S with H does not cause any assignment to any variable of S. A generator is a mature goal that is not a test. The proposed computation rule ensures that a mature subgoal does not change its status (test or generator) as a result of subsequent derivations. However, a mature goal may become a waiting goal or change status when the clause for its next call changes.

For example, let L and R be leftmost and nonleftmost variables of subgoal $\mathsf{Apnd}([\,], L, R)$, respectively. The subgoal is mature (generator) with respect to head $\mathsf{Apnd}([\,], X, X)$ since a most general unifier for the case is

$$L := R, \qquad X := R.$$

4. Conclusions

A preliminary implementation of the PROLOG interpreter using the proposed computation rule is described in [3,4]. The implementation maintains, for each variable in the goal, a list of subgoals waiting on an assignment to the variable. A subgoal spends time executing function wait, from its creation to maturity, equal to the time required by its call in a conventional interpreter [6]. However, a backtrack may delete some of the subgoals, thus losing the time spent on them. The main tradeoff, therefore, is between the reduction in the number of calls and the time lost due to backtrack. Since the proposed computation rule substantially reduces the need for backtrack, it must perform better than its conventional counterpart on nondeterministic programs. Experiments with our implementation suggest that the per-call time is about three times that of a conventional interpreter. We expect that a careful reprogramming can reduce this factor appreciably.

Table 1 compares the performance of an interpreter based on the proposed method with that of a conventional interpreter. The data suggests that the scheme benefits nondeterministic programs without adversely affecting deterministic programs.

Acknowledgment

We gratefully acknowledge the suggestions made by David Gries, Christian Codognet, Philippe Codognet and Vipin Kumar to improve the clarity of this paper.

References

[1] M. Bruynooghe and L.M. Pereira, Deductive revision by intelligent backtracking, in: J. Campbell, ed., *Implementation of Prolog* (Ellis Horwood, Ltd., Chichester, U.K., 1984) 194–215.

[2] W.F. Clocksin and C.S. Mellish, *Programming in Prolog* (Springer, Berlin/New York, 2nd ed., 1984).

[3] A. Kumar, *A Look-Ahead Interpreter for Prolog*, M. Tech. Thesis, Dept. of Computer Science and Engineering, Indian Inst. of Technology, Kanpur, 1987.

[4] A. Kumar and V.M. Malhotra, A look-ahead interpreter for sequential Prolog and its implementation, in: K.V. Nori, ed., *Proc. 7th Conf. on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, Vol. 287 (Springer, Berlin, 1987) 470–484.

[5] J.W. Lloyd, *Foundations of Logic Programming* (Springer, Berlin, 1984).

[6] D.H.D. Warren, *Implementing Prolog — Compiling Logic Programs 1 and 2*, DAI Res. Repts. 39 and 40, Univ. of Edinburgh, 1977.