From: Proceedings of the Second International Symposium on Methodologies for Intelligent Systems.

THE "RID-REDUNDANT" PROCEDURE IN C-PROLOG

Huo-yan Chen * *and Benjamin W. Wah* **

Coordinated Science Laboratory and

Department of Electrical and Computer Engineering

University of Illinois at Urbana-Champaign

1101 W. Springfield Avenue

Urbana, Illinois 61801, U.S.A.

ABSTRACT

C-Prolog can conveniently be used for logical inferences on knowledge bases. However, as similar to many search methods using backward chaining, a large number of redundant computation may be produced in recursive calls. To overcome this problem, we have designed the "rid-redundant" procedure to rid all redundant computations in running multi-recursive procedures. Experimental results obtained for C-Prolog on the Vax 11/780 computer show that there is an order of magnitude improvement in the running time and solvable problem size.

KEYWORDS: Backward chaining, C-Prolog, multi-recursive procedures, redundancy.

1. INTRODUCTION

Previous research indicated that efficiency is one of the critical factors that affect the practicality of logic programming as a software engineering methodology [6,7,8,9,10]. One of the reasons that degrade efficiency is the many redundant computations that are often associated with a backward chaining search method. In this paper we study the elimination of redundant computations in C-Prolog programs.

In the C-Prolog interpreter, the inference engine always selects in each step the leftmost unevaluated subgoal in the current goal and matches clauses in the order of appearance in the program text [1,2,4]. According to this standard control strategy, there are a lot of repeated redundant computations in running recursive procedures.

Example 1. The C-Prolog program for p(N,A), which represents the predicate "N'th Fibonacci number is A," is as follows.

C1: p(1,1).
C2: p(2,1).
C3: p(N,A) :- N1 is N-1, N2 is N-2, p(N1,A1), p(N2,A2), A is A1+A2.

For simplicity, we use the following C3' instead of C3, although it is invalid in C-Prolog.

C3': p(N,A) :- p(N-1,A1), p(N-2,A2), A is A1+A2.

In evaluating p(5,A) using the C-Prolog interpreter, the following computations will be produced.

|?- p(5,A).

Invoking C3' under the unifier {5/N}, we get

---

* Visiting research scholar from Department of Computer Science, Jinan University, Guangzhou, P.R.China

** Research supported by National Aeronautics and Space Administration Grant NAG 1-613.

International Symposium on Methodologies for Intelligent Systems, Charlotte, North Carolina, U.S.A., October 1987.

? p(4,A1), p(3,A2), A is A1+A2.
Invoking C3' under the unifier {4/N, A1/A}, we get
? p(3,A11), p(2,A12), A1 is A11+A12, p(3,A2), A is A1+A2.

Here, p(3,X) is evaluated twice: the result of first time is in A11, and the result of second time is in A2. The number of repeated computations will increase greatly when N increases. This increases the running time and reduces the solvable problem size.

There are two methods to eliminate these redundant computations. First, the interpreter [11] can be revised, so it checks every subgoal before evaluating it to determine whether it has been evaluated before. The subgoal will be evaluated and results stored in a knowledge base if it has not been evaluated before. If results of the subgoal are already available in the knowledge base due to a previous evaluation, they will be accessed. The run-time overhead of this method is high, as every subgoal evaluated has to be checked. A second method is to determine by a pre-interpreter all possible redundant predicates before the program is executed. The determination of results already evaluated during execution is similar to that of the first method. Naughton has recently proved that detecting and removing redundant predicates is undecidable in general; that is, there is no algorithm that will find and remove all redundant predicates in an arbitrary C-Prolog program in effective time and space [12]. For a subclass of single-recursive procedures or linear-recursive procedures, which are procedures that only call themselves once in their definitions, he presented a linear-time algorithm to detect and remove redundant predicates [13]. In this paper we discuss the case of multi-recursive procedures. These are procedures that invoke themselves more than once in their definitions. As we are studying the simple case of procedures that call themselves recursively, the identification of possible redundant predicates is trivial, and the second method degenerates into the first method. The technique that we will use is similar to that of dominance in dynamic programming algorithms [5].

2. THE "RID-REDUNDANT" PROCEDURE

The "rid-redundant" procedure is written in C-Prolog and is called with the following form.

rid-redundant([Q,M,X],M0).

where Q, M, and M0 are its input parameters, and X is its output parameter. Q represents the given multi-recursive procedure, which must be defined in C-Prolog. M indicates the variable of recursion in the given multi-recursive procedure. M0 is the maximum of the variable M in the all facts of the definition of the given multi-recursive procedure. M0 is said to be cardinal of the given multi-recursive procedure. X is the output of the given multi-recursive procedure, which can be in any valid data type. For instance, Q, M, M0, and X in Example 1 are p, N, 2, and A, respectively.

The methodology to eliminate redundant computations is as follows.

(1) Design all clauses in the target program using C-Prolog.

(2) If one subgoal calls a clause with head p(N,A) that is multi-recursive and has cardinal N0, that is, p(N,A) invokes itself in its definition, then "p(N,A)" is replaced at the calling point by "rid-redundant([p,N,A],N0)."

(3) If the initial query invokes subgoal q(N,A), which is a multi-recursive procedure and has cardinal N0, then "q(N,A)" is replaced by "rid-redundant([q,N,A],N0)."

3. IMPLEMENTATION

The "rid-redundant" procedure, rid-redundant([Q,M,X],M0), is appended to the original C-Prolog program as a built-in predicate. It is written in C-Prolog itself. Before it is used, we need consult it. The basic algorithm is as follows.

Algorithm RID:

(1) Read the values of Q, M, M0. Suppose that the value of Q is q.
(2) If M ≤ M0, then call q(M,X) and exit.
(3) K := M0+1.
(4) If K ≤ M, then
    (a) Call q(K,Xk).
    (b) Insert q(K,Xk) to the knowledge base as the current first clause for subgoal q.
    (c) K := K+1.
    (d) Go to Step (4).
(5) Call q(M,X).

(6) K := M.
(7) If K > M0, then
  (a) Erase q(K,Xk) from the knowledge base.
  (b) K := K-1.
  (c) Go to (7).
(8) Exit.

The C-Prolog program for the "rid-redundant" procedure is as follows.

```
rid-redundant([Q,M,X],M0) :- M=<M0, F=..[Q,M,X], call(F).
rid-redundant([Q,M,X],M0) :- M>M0, K is M0+1, add([Q,K,Xk],M),
      F=..[Q,M,X], call(F), delete([Q,M,Xm],M0).
add([Q,K,Xk],M) :- K=<M, Y=..[Q,K,Xk], call(Y), asserta(Y), K1 is K+1, !, add([Q,K1,Xk1],M).
add([Q,K,Xk],M) :- K>M.
delete([Q,K,Xk],M0) :- K>M0, Z=..[Q,K,Xk], retract(Z), K0 is K-1, !, delete([Q,K0,Xk0],M0).
delete([Q,K,Xk],M0) :- K=<M0.
```

4. EXPERIMENTAL RESULTS

We have implemented and tested the above "rid-redundant" procedure in C-Prolog running on a VAX 11/780 computer. In our experiments, the following statements were added to print the CPU time used in the computation of p(N,A).

```
|?- T1 is cputime, p(N,A), T2 is cputime.
```

where T1 is the CPU time used in seconds before running p(N,A) since C-Prolog was started, and T2 is the CPU time used after running p(N,A). The difference between T2 and T1 is the CPU time of running p(N,A).

In evaluating p(17,A) in Example 1, 6.1 seconds were needed, while 0.3333 seconds were needed to run rid-redundant([p,17,A],2). The speed is, therefore, increased by 18.5 times. On the other hand, running p(18,A) results in "out of local stack," but rid-redundant([p,184,A],2) can be evaluated successfully. Hence, the solvable problem size is increased by 184/17=10.8 times.

The following examples were also tried.

Example 2.

```
q(1,1).
q(2,1).
q(3,1).
q(N,B) :- N1 is N-1, N2 is N-2, N3 is N-3, q(N1,B1), q(N2,B2), q(N3,B3), B is B1+B2+B3.
```

The CPU time of running q(15,B) is 6.2666 seconds, but the CPU time of running rid-redundant([q,15,B],3) is only 0.3 seconds. Hence the running speed is increased by 20.9 times. On the other hand, running q(16,B) results in out of local stack, but rid-redundant([q,146,B],3) can be successfully run. So the solvable problem size is increased by 146/15=9.7 times.

Example 3.

```
s(1,1).
s(2,10).
s(N,D) :- N1 is N-1, N2 is N-2, s(N1,D1), s(N2,D2), D is (D1+D2)/2.
```

The CPU time of running s(17,C) is 6.8666 seconds, while the CPU time of running rid-redundant([s,17,C],2) is only 0.35 seconds. Hence, the running speed is increased by 19.6 times. On the other hand, running s(18,C) results in out of local stack, but rid-redundant([s,1000,C],2) can be run, so the solvable problem size is increased by 1000/17=58.8 times.

Example 4.

```
r(N,A) :- p(N,B), q(N,C), A is B+C.
r1(N,A) :- rid-redundant([p,N,B],2), rid-redundant([q,N,C],3), A is B+C.
```

where p, q are the same as in Examples 1 and 2, respectively. In our experiment, the CPU time of running r(14,D) is 4.88 seconds, but the CPU time of running r1(14,D) is only 0.533 seconds. The running speed is increased by 9.15 times. On the other hand, running r(15,D) results in out of local stack, but r1(146,D) can run well, so the solvable problem size is increased by 146/14=10.4 times.

5. DISCUSSION

Sometimes, the calling of non-multi-recursive procedures can also result in repeated computations. An example is shown below.

Example 5.

```
f(c).
f(a).
g(b).
s(a,b).
p(X,Y) :- q(X,Y), t(X,Y).
q(X,Y) :- f(X), g(Y).
t(X,Y) :- f(X), s(X,Y).
```

In invoking p(X,Y), the current goal will become

```
? f(X), g(Y), f(X), s(X,Y).
```

which results in f(X) evaluated twice.

Sometime, redundant computations produced by non-multi-recursive procedures in C-Prolog may be useful, and they should not be eliminated. The following example illustrates this point.

Example 6.

```
g(a).
t(a).
t(b).
s(c).
p(X,Y) :- q(X), r(Y), g(X).
q(X) :- g(X), t(X).
r(Y) :- asserta(g(b)), retract(g(a)), s(Y).
```

If we query the initial goal

```
!?- p(X,Y).
```

then at some moment in running this program, the current goal will become

```
? g(X),t(X),r(Y),g(X).
```

At this moment, the Prolog interpreter will have to solve g(X) twice, but we can not rid any one of them.

Suppose that G1 and G2 are two subgoals of goal G; suppose further that G1 and G2 are the same predicate, if the built-in predicate "asserts" ("asserta," "assertz"), or "retract" appears between G1 and G2 in G, or another subgoal Gi between G1 and G2 calls "assert" ("asserta," "assertz") or "retract" directly or indirectly, then, in general, we can not eliminate either G1 or G2, although they are repeated.

6. CONCLUSIONS

The above experimental results show that the "rid-redundant" procedure can increase the running speed by more than ten times, and can increase the solvable problem size by about ten times for multi-recursive procedures in C-Prolog we have tested. For non-multi-recursive procedure in C-Prolog, the "rid-redundant" procedure does not have any side effects.

REFERENCES

[1] C. J. Hogger, *Introduction to Logic Programming*, Academic Press, London, 1984.

[2] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, NY, 1981.

[3] Fernando Pereira, *C-Prolog User's Manual*, Version 1.3, EdCAAD, 1983.

[4] H. Gallaire, "Logic Programming: Further Developments", *IEEE Symp. on Logic Programming*, 1985.

[5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.

[6] R. E. Davis, "Logic Programming and Prolog: A Tutorial," *IEEE Software*, vol. 2, no. 5, Sept. 1985.

[7] M. R. Genesereth and M. L. Ginsberg, "Logic Programming," *CACM*, vol. 28, no. 9, Sept. 1985.

[8] R. Kowalski, "Predicate Logic as Programming Language," *Proc. IFIP*, 1974.

[9] R. Kowalski, "Algorithm=Logic+Control," *CACM*, vol. 22, no. 7, July 1979.

[10] T. Murata and D. Zhang, "A High-Level Petri Net Model for Parallel Interpretation of Logic Programs," *Proc. Int'l Conf. on Computer Languages*, 1986.

[11] M. H. van Emden, "An Interpreting Algorithm for Prolog Programs," *Proc. First Int'l Conf. on Logic Programming*, 1982.

[12] J. F. Naughton, "Redundancy in Function-Free Recursive Rules," *IEEE Symp. on Logic Programming*, 1986.

[13] J. F. Naughton, *Optimizing Function-Free Recursive Inference Rules*, Tech. Rep. STAN-CS-86-1114, Department of Computer Science, Stanford University, 1986.