Preventing Infinite Looping in Prolog

J. L. Han
School of Information Technology
University College of Southern Queensland
Toowoomba, Queensland Australia 4350

Abstract

This paper proposes a method to modify Prolog compilers so that infinite looping can often be avoided in recursive Prolog programs. The study is carried out in the context of deductive databases. Termination of recursive query processing here is related to boundedness: uniform boundedness and extensional boundedness. Limitations of this method are briefly discussed.

1 Introduction

Recursions are powerful but may cause infinite looping and redundancy. In this article, a method is proposed to modify Prolog compilers so that infinite looping can often (but not always) be avoided. To make implementation simple and to reduce computational complexity, modifications are kept to a minimum.

Let us first consider two simple examples. The same-generation relation is defined by the parent relation plus the following two logic rules:

\[ \text{sg}(X,Z) \]
\[ \text{sg}(Y,Z) : - \text{sg}(Z_1,Z_2), \text{par}(X,Z_1), \text{par}(Y,Z_2). \]  

(1)

(2)

The above Prolog program, as it is written, falls into infinite looping for query \( ?- \text{sg}(X,Y) \). There is, however, a programming trick which can prevent infinite looping for this example. Rewrite rule (2) as:

\[ \text{sg}(X,Y) : - \text{par}(X,Z_1),\text{par}(Y,Z_2), \text{sg}(Z_1,Z_2). \]  

(2')

then infinite looping will not occur unless the data are cyclic [1] for the parent relation.

Another example concerns the married relation. Suppose it includes a tuple \( \text{married}(joe, carol) \). Since the order of arguments is significant in a predicate, query \( ?- \text{married}(carol, joe) \) would be false. One may add the following simple recursive rule:

\[ \text{married}(X,Y) : - \text{married}(Y,X). \]  

(3)

However, a query such as \( ?- \text{married}(X,Y) \), even \( ?- \text{married}(joe, carol) \), falls into infinite looping under backtracking. To rectify this problem, one may keep the married relation unchanged and add the following two rules:

\[ \text{couple}(X,Y) : - \text{married}(X,Y). \]  

(4)

\[ \text{couple}(X,Y) : - \text{married}(Y,X). \]  

(5)

The above two examples show that infinite looping can often be avoided in Prolog by careful program-

ming. But this needs good programming skills. A Prolog compiler may be modified so that infinite loop-
ing can be avoided in many recursive programs.

2 Boundedness

Our results are based on deductive database [6] research. A deductive database is a relational database integrated with a set of logic rules. A Prolog program may be regarded as a deductive database. If a de-
ductive database has no function symbols, then it is a Datalog program. This article focuses on Datalog programs although the method is applicable to some general programs.

The relationship among predicates may be expressed by the dependency graph [6]. Each predicate in a Prolog program is represented by a node in the dependency graph. An arc is drawn from node \( A \) to node \( B \) whenever \( A \) appears in the rule body which is headed by \( B \). If a program is recursive, then its de-
dependency graph has at least one cycle. A rule is linear recursive if the head predicate appears once and only once in the rule body and the predicate is the recur-
sive predicate. A single rule recursion is a recursion with only one linear recursive rule and a set of non-
recursive rules. (It has one and only one single-node cycle in its dependency graph.) We consider mainly function-free single rule recursions.

We partition a single rule recursion into four parts: (1) the linear recursive rule (RR), (2) the initialization conditions (IC), (3) the extensional database (EDB), and (4) non-recursive rules (NR). The IC consists of facts which match the recursive predicate and/or non-
recursive rules whose heads match the recursive predic-
tive, such as rule (1). All other facts are in the EDB and other non-recursive rules are in the NR. Non-
recursive predicates are EDB predicates.

The meaning of logic rules may be computed iteratively by a bottom-up evaluation [7, 6]. If the program does not have function symbols (Datalog), then such an evaluation terminates after all solutions are ob-
tained. The number of recursive calls needed to obtain all solutions is the processing bound. The processing bound normally depends on the RR, IC, EDB, and the query. For some recursions, the bound can be decided by the RR alone. Such boundedness is called uniform boundedness [4, 5]. If a bound is decided by the RR and EDB then it is an extensional bound. The first example in the introduction is related to extensional boundedness while the second example is related to
uniform boundedness.

Uniform boundedness may be studied in the α-graph [4] (or equivalently the variable-predicate (V-P) graph [3]) or the argument-variable graph [5]. Ioannidis [4] gave a test on uniform boundedness:

Theorem 1. A recursive rule subject to restrictions below is uniformly bounded if there is no non-zero weight cycle in its corresponding α-graph. The bound is the maximum weight of all partial paths minus 1.

The restrictions are (1) linear recursive, (2) function-free, (3) no constants, (4) no repeated variables in the rule head, and (5) no degenerate cycles. In the 0th V-P graph [3], a cycle is degenerate if it contains only directed edges. Rule (3) in the introduction is represented by a degenerate cycle of weight 2.

Using expansions [4, 5, 3], it is not difficult to prove the following theorem.

Theorem 2. A linear recursive function-free rule represented by an isolated degenerate cycle in the 0th V-P subgraph is uniformly bounded. The bound is \( w - 2 \) if \( w > 1 \) and \( 0 \) if \( w = 1 \), where \( w \) is the weight of the cycle.

A 0th V-P subgraph may have several uniformly bounded components with different bounds. It is easy to show that the bound for the corresponding rule should be the maximum of all bounds.

Extensional boundedness is determined by the RR, EDB, and possibly the IC or query. The predicate-goal (P-G) graph [2] transforms recursive query processing into a graph matching problem. The meaning of a linear recursive rule is well defined in a P-G graph yet no procedure detail is fixed and many algorithms are available. One can choose an algorithm for termination consideration or for query optimization.

A P-G graph contains many P-G subgraphs. EDB predicates are represented by solid edges. Solid edges may join (due to shared variables) to form E-chains. As the order of subgraph increases, the lengths of E-chains usually increase. The longer an E-chain, usually the more difficult to match it using the finite EDB. If an E-chain cannot be matched, then using expansion properties given in [3], it can be shown that an extensional bound is reached.

3 A processing strategy

First, a program is preprocessed. (1) Use the dependency graph [6] to find recursive rules and recursive predicates. (2) Apply the above two theorems if they are applicable and obtain the uniform bounds. (3) Determine the IC for each cycle of the dependency graph.

Query processing proceeds as usual [7] until a recursive predicate is encountered. (1) If the recursive predicate appears in a rule body, move it back to the end of the rule body. (2) If the recursive predicate is defined by a uniformly bounded recursive rule, then set up a counter to record the number of recursive calls. The counter is initialized to 0 when the recursive rule is invoked. The counter’s value is increased by one each time this recursive rule is applied. The initial recursive predicate fails when the counter’s value is greater than the uniform bound. (3) When there are several rules to choose, non-recursive rules take precedence over recursive rules. (4) When there are several predicates in the rule body, EDB predicates take precedence over recursive predicates. Sometimes, due to uninstantiated variables in arithmetic and comparison operators evaluation of EDB predicates must be delayed and the precedence rules need to be overridden.

The computation complexity of the above processing strategy is not too high. The dependency graph is simple. There is a polynomial algorithm for Theorem 1 [4] while Theorem 2 is simple.

Delay of recursive calls may work for non-linear recursive rules and rules with functions. At least this method can do no worse than the original evaluation. But many issues remain to be addressed: (1) non-linear recursive rules, (2) mutually recursive rules and multi-level recursions, (3) functions, (4) degenerate cycles integrated with other edges, (5) repeated variables and constants in recursive rules, (6) arithmetic and comparison operators, (7) cyclic data, (8) limitations of uniform and extensional boundedness, and (9) query optimization.

Issue (8) is difficult [5]. Neither uniform boundedness nor extensional boundedness is sufficient to deal with all recursions.

If functions are allowed then a different kind of problem may occur. For example, consider

\[ r(a). \]

(6)

\[ r(f(x)) \rightarrow r(x). \]

(7)

Repeated recursive calls are not infinite looping here. Rather, the program has infinite answers and each recursive call produces one more answer.

References


