LOOP DETECTION IN PROLOG IN THE CASE OF POSSIBLE BACKTRACKING

(extended abstract)

Dimiter Skordev
Sofia University, Sofia, Bulgaria
Igor Đurđanović
Paderborn University, Paderborn, Germany

1 Introduction

A method will be proposed for the detection of some periodic loops during the execution of Prolog programs, and an implementation of this method will be described.[1] The method is an extension of another one presented in the paper [3]. Namely, that method was intended for the detection of loops only during depth-first search, whereas the new one will work essentially in the same way in that case, but the combination of depth-first search with backtracking will be no more an obstacle.[2]

As in [3], we shall consider only Prolog programs consisting of Horn clauses and goals consisting of atomic formulas. For describing our loop detection method and specifying the class of the detected loops, we use a mathematical model of Prolog execution with backtracking. The model takes into account only such features of the execution process which are of interest for our purpose; due to this, it is much simpler than, for example, the D. H. D. Warren Prolog Engine (cf. [1], pp. 447-471).

From now on, we suppose a Prolog program $\mathcal{P}$ is given. Let $C_1, \ldots, C_m$ be the clauses of this program. We shall consider objects called marked goals. By definition, a *marked goal* is an ordered pair $<k, G>$, where k is some of the numbers $0, 1, \ldots, m$, and G is an ordinary goal (if G is the goal consisting of the atomic formulas $A_1, \ldots, A_l$, then $<k, G>$ will be denoted also by $k - A_1, \ldots, A_l.$). Roughly speaking, the number k is intended to forbid using clauses C_i with $i \leq k$ for a resolution with G. The number of the atomic formulas in G will be called *the length of* $<k, G>$. The *beginnings of* $<k, G>$ are, by definition, the marked goals $<k, G'>$, where G' is a beginning of G. Two marked goals $<k, G>$ and $<k', G'>$ will be called to be *essentially the same* if $k = k'$, and G coincides with G' up to some renaming of variables.

Any finite sequence of marked goals will be called an *execution state* (such a sequence $(M_1, M_2, \ldots, M_d)$ will be denoted also by $M_1\, M_2\, \ldots\, M_d$, and thus the one-term sequence (M_1) will be identified with its only member M_1). Intuitively speaking, an execution state is intended to represent some current content of a stack needed for the execution of $\mathcal{P}$. We assume that the direction from top to bottom in the stack is represented by the one from left

[1]The method has been designed by the first author during a stay at the Paderborn University, and the implementation is done by the second author. The research of the first author has been supported by a DAAD grant and by the Bulgarian Ministry of Science and Higher Education - Contract No. MM 43, 1991.

[2]For detecting loops only during depth-first search, we have now a simpler method, which is also implemented.

to right in the sequence. The first member of a non-empty execution state will be called its *top goal* (marked goals will be sometime called simply goals). The number of the members of an execution state will be called *the depth* of this state. Two execution states will be called to be *essentially the same* if they have one and the same depth, and any member of the first state is essentially the same as the corresponding member of the second state.

An execution state will be called *terminal* if either it has depth 0, or it has a top goal with length 0. Let $M_1\, M_2\, \ldots\, M_d$ be a non-terminal execution state with $M_1 = <k, G>$. Another execution state, called *the successor of* $M_1\, M_2\, \ldots\, M_d$, will be defined in the following way. If there is a clause C_i with $i > k$ and with a head unifiable with the leftmost formula of G, then the successor of $M_1\, M_2\, \ldots\, M_d$ is $<0, G'> <j, G> M_2\, \ldots\, M_d$, where C_j is the first one among the clauses C_i with the above properties, and G' is obtained from G and C_j by resolution upon the leftmost formula of G. Otherwise the successor of $M_1\, M_2\, \ldots\, M_d$ is $M_2\, \ldots\, M_d$. In the first of these cases, G' could contain variables whose choice is not uniquely determined, but we overcome this problem by identifying any two execution states which are essentially the same.

A finite or infinite sequence of execution states will be called an *execution path* if the following condition is satisfied: whenever an execution state E is a member of the sequence, then there is a next member of this sequence if and only if E is non-terminal, and in this case the successor of E is the next member. Clearly, for any given execution state E_0, there is, under the above mentioned identification of execution states, exactly one execution path with initial member E_0. If a query is given, then *the execution of* $\mathcal{P}$ *on this query is represented* by the execution path with initial member $<0, G_0>$, where G_0 is the goal from the query.

2 The detection method

A marked goal M will be called *strongly cyclic* if the execution path with initial member M contains some further member with a top goal having a beginning essentially the same as M. A marked goal will be called *cyclic* if some of its beginnings is strongly cyclic. We say that *the execution of* $\mathcal{P}$ *on a given query enters into a cyclic loop* if the execution path representing this execution contains some member with a cyclic top goal (the execution path is surely infinite in this case). Our method detects exactly such kind of loops. A brief description of the method follows.

First of all, a computable strictly increasing infinite sequence $\tau_0 < \tau_1 < \tau_2 < \tau_3 < \ldots$ of natural numbers must be chosen such that the set of the differences $\tau_{i+1} - \tau_i$ is unbounded. The set of the members of this sequence will be further denoted by $\mathcal{T}$.

The application of the method can be characterized as carrying out execution steps accompanied by certain loop detection activities. Namely, the depth and the top goal of the current execution state have to be saved at appropriate moments and to be used further for the loop detection during a certain period of time, and this must be repeated until possibly a terminal execution state is reached or a loop is detected. The moments for saving are the ones belonging to $\mathcal{T}$ and those ones, at which the depth of the current execution state becomes less than the saved depth. For the needs of the loop detection, at any moment later than τ_0, a certain natural number p is considered (*the length of the still passive part of the saved*

Program:

```
p(a,a).
p(X,X) :- p(Y,X).
q(b).
```

Query:

```
?- p(U,U),q(U).
```

Path:

```
<0, 0, 0-.,              0, 0-p(U,U),q(U).>,
<1, 1, 0-p(U,U),q(U).,   1, 0-q(a).1-p(U,U),q(U).>,
<2, 2, 0-q(a).,          0, 1-p(U,U),q(U).>,
<3, 1, 1-p(U,U),q(U).,   1, 0-p(Y,U),q(U).2-p(U,U),q(U).>,
<4, 1, 1-p(U,U),q(U).,   1, 0-q(a). 1-p(Y,U),q(U). 2-p(U,U),q(U).>,
<5, 1, 1-p(U,U),q(U).,   0, 1-p(Y,U),q(U).2-p(U,U),q(U).>,
<6, 1, 1-p(U,U),q(U).,   0, 0-p(Y1,Y),q(Y).2-p(Y,U),q(U).2-p(U,U),q(U).>,
<7, 3, 0-p(Y1,Y),q(Y)., 1, 0-q(a).1-p(Y1,Y),q(Y).2-p(Y,U),q(U).2-p(U,U),q(U).>,
<8, 3, 0-p(Y1,Y),q(Y)., 0, 1-p(Y1,Y),q(Y).2-p(Y,U),q(U).2-p(U,U),q(U).>,
<9, 3, 0-p(Y1,Y),q(Y)., 0, 0-p(Y2,Y1),q(Y1).2-p(Y1,Y),q(Y).2-p(Y,U),q(U).
                                                        2-p(U,U),q(U).>.
```

Figure 1: Execution-detection path

goal). It is computed in the following way: at the next moment after a saving is done, the value of p is equal to the length of the saved top goal minus 1; at any further step till the next saving, the value of p decreases by 1, if the top goal of the current execution state before the step has a length equal to p, and does not change otherwise. The number p is always non-negative and less than the length of the saved top goal. The beginning of that goal with a length complementary to p is called *the already activated part of the saved goal*. *A loop is detected* at a moment later than τ_0 if the execution state at this moment has a depth not less than the saved one, the top goal of this state has a length not less than the length of the saved goal, and there is a beginning of this top goal essentially the same as the already activated part of the saved goal.

The precise description of the detection method uses quintuples $<t, d, M, p, E>$, where M is a marked goal (the previously saved goal), t, d, p are natural numbers (moment of time, previously saved depth, length of the still passive part of the saved goal), and E is an execution state (the execution state at moment t). The application of the method to the execution of $\mathcal{P}$ on a given query is represented by a sequence of such quintuples. We call this sequence *the execution-detection path of* $\mathcal{P}$ *on the given query*. An example of such an execution-detection path is shown on Figure 1. It is assumed that $\tau_0 = 0$, $\tau_1 = 1$, $\tau_2 = 6$, $\tau_3 \geq 9$ in this example. The second, the third and the fourth component of the initial member of the path are written only by technical reasons, since there is no previous saving at the moment 0. Saving is done at moments 0, 1, 2 and 6. A loop is detected at moment 9.

Details concerning the correctness proof for the proposed loop detection method will be presented in the complete paper.

3 Implementation

The dilemma was whether to implement the algorithm in interpreter code or in Prolog. The first solution obviously offers speed and complete control of Prolog machine, but it requests knowledge of local Prolog implementation, which is usually hidden from end user. For the sake of simplicity and generality our implementation is done in Prolog itself. A small (and incomplete) interpreter of Prolog in Prolog is writen, which is capable of controling backtracking, unification and SLD-resolution.

The lack of global variables is overcomed in standard (unfortunately expensive in time and space) way by using built-in Prolog predicates: `assert` and `retract`, thus avoiding incompatibility with different Prolog implementations and their solutions to this problem. Although the final implementation is to be done by using global variables for the sake of speed and space.

The controling of backtracking was done by introducing clause numbers in process of head-unifying, which enables us to request (on backtracking) that the number of clause which is head-unifiable with first subgoal to be greater than one used before as shown in Figure 2.

Internaly, goal as a conjuction of subgoals, is saved as a list of subgoals which expands (or shrinks) due to SLD-resolution applied on first subgoal of current goal. All clauses of inspected program are slightly modified to make a database of clauses (see Figure 3 which shows a simple example) with all necessary informations like: clause number, head and body of clause and length of clause which is determined in read-modify-assert process of loading inspected program.

```
'.sld'(Unify_c, Goal_c, First, SLD, SLD_1) :-
      '.SLD'(Goal_c, First, SLD, SLD_1),
      Unify_c < Goal_c, !.
'.sld'(_, 0, _, _, _).
```

Figure 2: Control of backtracking and SLD-resolution

Problem of comparing active parts of saved goal and current goal with renaming of variables is solved by flattening complex expressions in order to reach all variables (even those burried deep in structures) and then binding them parallely to unique term (this make us capable of distinguishing between variables). Unbinding is forced by backtracking, but with saving result of comparison.

The whole implementation took about 170 lines of Prolog code which can be even more reduced by using local implementations of global variables, predicate for generating new terms (atoms), and omitting all outputs of program (currently present only for demonstrating work of algorithm). Implementation was done in C-Prolog on Sun under Unix.

4 Acknowledgments

We are grateful to Prof. Dr. H. Kleine Büning and to Dr. Th. Lettmann for a useful discussion concerning the implementation problem for another our loop detection method

Modification algorithm:

```
'.and2list'( (F,R), [F|RL], N ) :- !, '.and2list'(R,RL,RN), N is RN + 1, !.
'.and2list'( G, [G], 1 ).

'.modify'( N, (H :- B), '.SLD'(N, H, BL, L) ) :- !, '.and2list'(B,BL,L).
'.modify'( N,  C      , '.SLD'(N, C, [], 0) ).
```

Program to be modified:

```
p(a).
p(X) :- q(X),p(X).
q(b).
q(a).
```

Program as a database:

```
'.SLD'(1, p(a), [], 0).
'.SLD'(2, p(X), [q(X),p(X)], 2).
'.SLD'(3, q(b), [], 0).
'.SLD'(4, q(a), [], 0).
```

Figure 3: Modification example

(treating the case of depth-first search). The discussion helped essentially to implement that method. The success in its implementation strongly stimulated us for the design and the implementation of the present more complicated method.

References

[1] D. Maier, D. S. Warren, Computing with Logic. Logic Programming with Prolog, Benjamin/Cummings Publ. Company, Menlo Park, 1988.

[2] D. Skordev, An extremal problem concerning the detection of cyclic loops, *C. R. Acad. Bulgare Sci.*, **40**, No. 10, 1987, 5-8.

[3] D. Skordev, On the detection of some periodic loops during the execution of Prolog programs, *Banach Center Publications*, Warsaw (to appear).