A FURTHER NOTE ON LOOPING IN PROLOG

Michael A. Covington

Advanced Computational Methods Center
The University of Georgia
Athens, Georgia 30602

0. Abstract

In an earlier paper (Covington 1984) I proposed that looping in Prolog should be prevented by blocking all derivations in which the current goal matches a higher goal. This is too strong a constraint; it causes exhaustive searches to be incomplete. A better approach is to block all derivations in which the current goal matches a higher goal and is about to be evaluated using the same rule as was used to evaluate the higher goal. This is sufficient to block looping, and produces otherwise correct behavior, in all of the cases mentioned in the earlier paper.

1. Loop checking

As noted in Covington (1984), the following types of rules produce infinitely looping derivations:

(Symmetry:) `f(x,y) if f(y,x).`

(Transitivity:) `f(x,y) if f(x,z) and f(z,y).`

In each of these cases, the rule applies recursively to the subgoal that it invokes. The same thing happens when a pair of rules expresses a biconditional:

(Biconditional:)
```
f(x) if g(x).
g(x) if f(x).
```

Here each rule applies to the subgoal invoked by the other. Loops are not limited to these easily recognizable rule configurations; a loop can also occur indirectly through an arbitrarily long series of rules, such as:

```
f(x) if g(x).
g(x) if h(x).
h(x) if k(x).
k(x) if f(x).
```

Symmetric and transitive relations and biconditional implications are common in human thought, and the inability to express them successfully is a serious limitation of Prolog. In order to handle them correctly, the Prolog inference engine should block all the looping derivations, while allowing all other derivations to proceed normally.

The original loop checking algorithm was based on the following assumption:

(Assumption 1) The actions required to evaluate a Prolog goal depend only on the current goal and the current rule set.

Thus (my argument ran), if a given goal invokes a copy of itself, then that copy will also invoke a copy of itself, ad infinitum. This conclusion obviously does not apply in the case of goals that have side effects such that Assumption 1 does not apply -- for example, goals that alter the rule set or perform input from a file. The loop checking algorithm is therefore:

(Algorithm Alpha) Fail any goal that exactly matches a higher goal, provided that neither it nor any intervening goal has side effects.

(Goals with side effects have to be allowed to go through, since there is no way to predict whether or not they will loop.)

Algorithm Alpha can be performed in nearly linear time by having each goal leave its signature in a hash table when it is invoked. If a goal finds that its signature has already been marked, it triggers a search for a matching higher goal.

2. Inadequacy of Algorithm Alpha

Consider the following Prolog rule set:

```
[1]  s(a, b).
[2]  s(b, c).
[3]  s(c, d).
[4]  s(x, y) if s(x, z) and s(z, y).
```

The goal **s(a,d)** succeeds from this rule set with or without loop checking, as follows:

(Derivation 1)

```
Goal:     s(a, d)
Rule 4:   s(a, d) if s(a, z) and s(z, d).
Goal:     s(a, z) and s(z, d)
Rule 1:   s(a, b).   z = b.
Goal:     s(b, d).
Rule 4:   s(b, d) if s(b, z1) and s(z1, d).
Goal:     s(b, z1) and s(z1, d)
Rule 2:   s(b, c).   z1 = c.
Goal:     s(c, d)
Rule 3:   s(c, d).   Success.
```

If the computer is asked for all **x** and **y** such that **s(x,y)** succeeds, it should generate **s(a,d)** as one of the answers. Without loop checking, it does so, as follows:

(Derivation 2)

```
Goal:     s(x, y)
Rule 4:   s(x, y) if s(x, z) and s(z, y).
Goal:     s(x, z) and s(z, y)
Rule 1:   s(a, b).   x = a,  z = b.
Goal:     s(b, y)
Rule 4:   s(b, y) if s(b, z1) and s(z1, y).
Goal:     s(b, z1) and s(z1, y).
Rule 2:   s(b, c).   z1 = c.
Goal:     s(c, y).
Rule 3:   s(c, d).   y = d.   Success.
```

But it never generates **s(b,d)**; before doing so, it gets into a loop in a derivation that begins like this one, but then applies rule 4 over and over instead of concluding with rules 2 and 3 (see Covington 1984).

With full loop checking as proposed in Covington 1984, this loop is prevented and it is possible to get **s(b,d)**, but Derivation 2 itself does not go through, and it is impossible to get **s(a,d)**. This is because Derivation 2 contains two instances (underlined) of the current goal matching a higher goal. (Remember that all uninstantiated variables are equivalent to each other.)

The problem is that Assumption 1 does not hold during an exhaustive search. The reason for this is that, in order to make the search exhaustive, rules are blocked off one by one on successive attempts to reach a solution, so the rule set does not remain constant. This suggests:

(Algorithm Beta) Fail any goal that exactly matches a higher goal **and has the same set of rules available**, provided that neither it nor any intervening goal has side effects.

Keeping track of the entire rule set for each subgoal obviously involves an unmanageable amount of data. The algorithm can, however, be simplified by noting that the only rules that are trimmed away are those that would have been used first if they were present. Thus:

(Algorithm Gamma) Fail any goal that exactly matches a higher goal **and is going to be evaluated by the same rule**, provided that neither it nor any intervening goal has side effects.

This can be implemented in the same way as Algorithm Alpha, by including the rule in the information to be hashed. It appears to solve the problem completely.

Reference

Covington, M. A. 1984. Eliminating unwanted loops in Prolog. *ACM SIGPLAN Notices* 20.1:20-26.