EFFICIENT LOOP DETECTION IN PROLOG USING THE TORTOISE-AND-HARE TECHNIQUE

ALLEN VAN GELDER*

▷ A well-known problem with PROLOG-style interpreters that perform goal reduction is the possibility of entering an infinite recursion, due to a subgoal being "essentially the same" as one of its ancestors. This is informally called a "loop". We describe the tortoise-and-hare technique for detecting such loops. This technique has low overhead: a constant amount of time and space per goal reduction step. Therefore it should be practical to incorporate into high-performance interpreters. We discuss the special considerations needed for correct implementation in an interpreter that uses tail-recursion optimization. The issue of what to do when a loop or potential loop has been detected has been investigated elsewhere. We review these results, and conclude that loop detection is probably more useful as a debugging tool than as an extension to the power of the language. ◁

1. GOAL-REDUCTION INTERPRETERS

Conventional PROLOG interpreters work by goal reduction, as described in [2] and elsewhere. We shall sketch only the basic ideas here. Two important data structures maintained by the interpreter are the *goal stack* and *goal list*. Essentially the goal list contains *pending goals*, those whose solution is needed but not begun. The goal stack contains *open goals*, those whose solution is underway, and *closed goals*, which are solved. The *current subgoal* is the goal at the top of the goal stack, which is also the goal at the front of the goal list. We shall adopt a diagrammatic representation that makes the relationships visible without showing irrelevant implementation

Address correspondence to Allen Van Gelder, Department of Computer Science, Stanford University, Stanford, CA 94305-2085.

Received 25 March 1986.

*Supported by NSF grant IST-84-12791 and a grant of IBM Corp.

Level	Stack		
4	$a(2, V_4)$	$p(V_4, U_1)$	[]
3	$p(2, U_1)$	[]	
2	$a(1, 2)$		
1	$p(1, U_1)$	$b(U_1, 5)$	[]
0	$r(1, 5)$	[]	

FIGURE 1. Goal stack and goal list.

details. In our diagrams the goal stack is the leftmost column, and it grows upward. A "layer" is associated with each level in the stack; it includes the stack goal at that level, and possibly additional goals in columns to the right. A closed (solved) goal has nothing else in its layer. An open goal has additional goals in its layer; they occupy columns to the right of the stack, and are terminated with the "empty goal" marker []. (Possibly [] is the only additional goal in the layer.) The goal list is obtained by reading the goals top to bottom, left, to right, omitting those buried in the stack. We illustrate with an example.

Example 1.1. Suppose the interpreter is processing the logic program $\mathbf{P}_1$ with rules

R1: $p(X, Y) :- a(X, V), p(V, Y).$
R2: $p(X, Y) :- a(X, Y).$
R3: $r(X, Y) :- p(X, U), b(U, Y).$
R4: $a(1, 2).$
R5: $a(2, 1).$

followed by various other ground facts for $a(X, Y)$ and $b(X, Y)$. Symbols beginning with capital letters are variables, and ":-" is read as "if". When given the top level goal, $r(X, 5)$, after several steps the state of the computation is represented by the diagram in Figure 1.

This is a snapshot after four goal-reduction steps, using rules R3, R1, R4, and R1 again. In this picture, the goal list is

$$a(2, V_4), p(V_4, U_1), b(U_1, 5)$$

The *current subgoal* is the goal at the top of the stack and at the front of the list, $a(2, V_4)$ in this case. The goal $a(1, 2)$ at level 2 is closed; the other stack goals are open.

Briefly, a conventional PROLOG interpreter's next action when a new current subgoal is created is to find the first rule of the program whose head unifies with the current subgoal; the rule initially has all fresh variables. This rule's head is then unified with current subgoal. The unifying substitution applies to the rule's subgoals, the goal stack, and the goal list as well. The rule's subgoals then make up a new "layer" at the top of the diagram, as shown in Figure 2, where the first goal reduction uses rule R1 and the second uses R4.

When the current subgoal is [], this signifies that the goal immediately under it in the stack has *succeeded*, or been *solved*. In this case, the interpreter's action is illustrated in Figure 3.

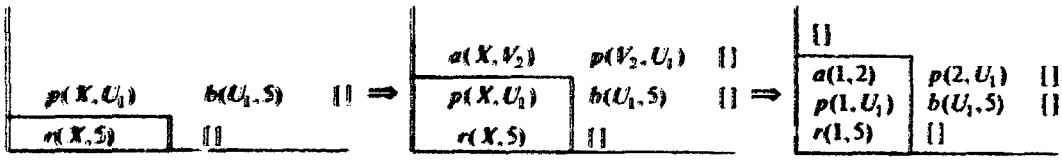

FIGURE 2. Goal reduction of $p(X, U_1)$ followed by goal reduction of $a(X, V_2)$.

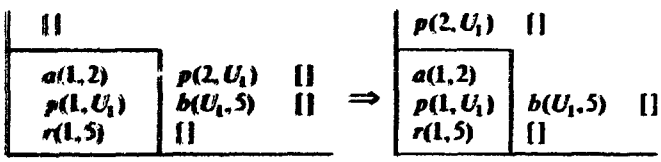

FIGURE 3. Action on solved goal $a(1, 2)$.

When no rule head unifies with the current subgoal, that goal is said to have *failed*; the interpreter backtracks to the state just prior to the previous goal reduction. All unifications done in that goal reduction are backed out. The "new" current subgoal is actually an old one that is now on top of the goal stack again. The next goal reduction will try only rules that are later in the program than the rule just backed out.

2. THE LOOPING PROBLEM

If we continue Example 1.1, the stack diagram evolves into Figure 4. It is evident that the interpreter is in an unproductive loop. It will continue to use rule R1 and grow the stack until it is externally aborted, usually by running out of space.

To avoid this behavior, the interpreter needs to "notice" when the new current subgoal is "essentially the same" as one already in the goal stack. We postpone a precise definition of this term. (Note that goals in the goal *list* are not relevant to loop detection.)

In this example the stack will repeat itself every four levels. However, the period could be much larger, depending on the data in a. Thus, in general, the interpreter would have to look arbitrarily deep into the stack in order to detect potential loops.

Another kind of problem behavior, which is not addressed in this paper, occurs when the interpreter repeatedly selects the same rule but with different (growing)

6	$a(1, V_6)$	$p(V_6, U_1)$	[]
5	$p(1, U_1)$	[]	
4	$a(2,1)$		
3	$p(2, U_1)$	[]	
2	$a(1,2)$		
1	$p(1, U_1)$	$b(U_1, 5)$	[]
0	$r(1,5)$	[]	

FIGURE 4. A looping computation.

arguments. For example, a rule like

$$lt(X, Y) :- lt(s(X), Y).$$

with X and Y instantiated could generate a series of goals

$$lt(X, Y) \Rightarrow lt(s(X), Y) \Rightarrow lt(s(s(X)), Y) \Rightarrow \cdots$$

which might or might not terminate, depending on the bindings. (See [4] for related examples and further discussion.) For our purposes, this is *not* considered a loop.

Although the examples we include have no function symbols for simplicity, the technique we shall present works equally well on programs with function symbols.

3. THE TORTOISE-AND-HARE TECHNIQUE

The goal stack routinely grows to a considerable height in PROLOG. Hence it would be very burdensome to check every stack entry against the new current subgoal after each goal reduction. The tortoise-and-hare technique is a simple scheme to avoid this problem. The idea comes from Knuth [3] (see Exercise 3.1.6b), who credits it to R. W. Floyd; the author first heard about it in a talk by Andrei Broder on an entirely unrelated problem.

The technique relies on the fact that a PROLOG interpreter will always handle variants of a given goal in exactly the same way (up to renaming variables). That is, it will always try clauses in the same order, and always reduce subgoals within a clause in the same order. Therefore, when it is in a loop as we have defined it, its behavior is perfectly periodic. This assumption could be violated if the problem had side effects, such as *assert* and *retract*, or even *read*, as shown in Example 4.2, later.

The tortoise-and-hare technique applied to PROLOG interpreters works by labeling alternate stack building steps *hops* and *walks*, beginning with a hop. We visualize the hare and tortoise moving up the goal stack as it is formed. The hare moves on both hopping and walking steps, and so stays at the top of the stack. The tortoise only moves on a walking step, and so is always about in the middle of the stack. After each step, the tortoise and hare compare their respective stack elements. If they are "essentially the same", a loop is detected.

To see why this works, assume the interpreter is in an infinite loop with cycle length k. That is, every new current subgoal is "essentially the same" as the one k deep in the stack. Eventually, the tortoise gets into the cyclic portion of the stack. Since the distance between the hare and the tortoise increases by one every two steps, at some later time, their distance is a multiple of k. At that time, they will compare subgoals that are "essentially the same", provided our definition of this property is transitive.

Example 3.1. In this example, two goals are "essentially the same" if they are variants of each other. Suppose the interpreter is processing the logic program $\mathbf{P}_2$ with rules

R1: $p(X, Y) :- q(X, U), p(U, Y).$
R2: $p(X, Y) :- a(X, Y).$
R3: $r(X, Y) :- p(X, U), b(U, Y).$
R4: $q(X, Y) :- r(X, Y).$

hare →	$p(X,U_3)$	$b(U_3,U_2)$	[]
	$r(X,U_2)$	[]	
tortoise →	$q(X,U_2)$	$p(U_2,U_1)$	[]
	$p(X,U_1)$	$b(U_1,6)$	[]
	$r(X,6)$	[]	

FIGURE 5. The tortoise and the hare.

followed by various ground facts for $a(X,Y)$ and $b(X,Y)$. The top level goal is $r(X,6)$. After four goal reductions the goal stack and goal list are as shown in Figure 5.

Although $p(X,U_3)$ is a variant of $p(X,U_1)$ in the stack, the interpreter does not "notice", because the tortoise is not there. However, the next step is a "hop". After it, the hare will be looking at $q(X,U_4)$, the tortoise will still be at $q(X,U_2)$, and the loop will be detected.

To complete the description of the technique, we need to specify what to do when a goal fails. Essentially we just undo the tortoise and hare motions. Thus whenever a layer created by a "hop" is removed, only the hare moves down; whenever a layer created by a "walk" is removed, both the tortoise and hare move down. A toggle can be maintained by the interpreter to record which type of step created the current top layer. In other words, "walk" is associated with even stack height, and "hop" with odd. Special considerations for tail-recursion optimization are covered later.

4. USES OF LOOP DETECTION

Since going into loops seems to be one of the things they do best, there have been various proposals to modify PROLOG interpreters one way or another to detect loops and automatically recover. However, it has been shown that trying to automatically recover and maintain completeness of solutions is essentially futile, even in the absence of function symbols [1,5], and the presence of functions can only make matters worse. Defining "essentially the same" to mean "variant", as we did in the previous example, catches many loops in practice, but the problem is that failing the new goal may cause correct solutions to be missed.

Example 4.1. Suppose in Example 3.1 that the following facts were given for a and b:

$a(0,1)$ $\quad$ $b(1,2)$
$a(2,3)$ $\quad$ $b(3,4)$
$a(5,6)$ $\quad$ $b(5,6)$

The top level goal is $r(X,6)$. After 5 goal reductions the loop is detected by the tortoise and hare. The current subgoal $q(X,U_4)$ is a variant of $q(X,U_2)$ in the stack. However, if the goal $q(X,U_4)$ is failed, the solution $r(0,6)$ will never be found.

A proposal sometimes seen is to check whether the rule about to be used on $q(X,U_4)$ is the same as was used on its earlier variant, $q(X,U_2)$; the goal is failed only if the same rule is to be used. This extra degree of caution does not save us, either.

But of course solution $r(0,6)$ will not be found if the interpreter is left to continue in the unproductive loop, either. Therefore the best course of action here is for the interpreter to give debugging control to the user, and to let the user analyze the problem.

Defining "essentially the same" to mean "exactly the same" results in many loops not getting caught; however, those that are caught apparently can be failed without losing any correct solutions. For any possible solution to the failed goal will be equally applicable to the goal in the stack that is the same as it [5]. But even here, we need a qualification: failing the repeated goal is safe only if the program is free of side effects.

Example 4.2. Silently failing even a repeated *ground* subgoal would be incorrect on the PROLOG program P_3 given by

$p(F) :-$ see(F), read(I), $s(I,0)$, seen.

$s($end_of_file, $N) :-$ write(N), nl.

$s(I, N) :- M$ is $N + I$, read(J), $s(J, M)$.

The interpreter makes a stack of ground subgoals of the form $s(I, N)$, where I is an integer just read from the file F, and N is the partial sum. Reading several zeros in a row causes apparently repeated subgoals. (A mixture of positive and negative numbers in the input can cause repeated subgoals also.) However, the state of the computation is actually changing, due to the input file being consumed. This side effect is not reflected in the goal stack.

As these examples show, it is not safe for a general purpose interpreter to take any action on its own initiative when it detects a loop. A complicated system of user options is probably not economically justified. Yet loop detection is easy to add to the interpreter, and hard (or at least expensive) for the user to simulate. Based on these considerations, an implementation along the following lines looks like a reasonable compromise:

The default mode for the interpreter is *checkloops*(2). In this mode, when a subgoal is detected to be a variant of one in the stack, the debugger is entered.

Another mode for the interpreter is *checkloops*(1). In this mode, when a subgoal is detected to be identical to one in the stack, the debugger is entered.

The third mode for the interpreter is *checkloops*(0). In this mode the tortoise and hare positions are kept up to date, but their subgoals are not automatically compared.

The predicate *checkloops*(I, P) causes mode I to be entered, and unifies P with the previous mode.

A built-in predicate *tortoise*(N) compares the "tortoise subgoal" with the top goal in the stack (other than *tortoise* itself, of course), and unifies N with 1 if they are identical, with 2 if they are variants, and with 3 otherwise.

This scheme allows the user to write a routine to handle repeated goals, or let the

interpreter do it, or skip it all together. However, caution is still necessary in environments where all solutions are sought, such as with *setof*, *bagof*, and *findall*.

Example 4.3. We are given a set of facts about a network in the form $n(X)$ and $a(X, Y)$, meaning "X is a node" and "X has an arc to Y", respectively. We want a procedure $p(X, Ys)$ that means "Ys is a list of all nodes Y such that X has a path of one or more arcs to Y". In short, for each X, Ys is its reachability set, possibly with duplicates. We would like to use *findall*, but the problem is that infinite recursions can occur. A tempting (but faulty) solution using the *tortoise* feature is $\mathbf{P}_4$:

$p(X, Ys) :- n(X)$, *checkloops*$(0, L)$, findall$(Y, s(X, Y), Ys)$, *checkloops*$(L, 0)$.
$s(X, Y) :- a(X, Y)$.
$s(X, Y) :-$ *tortoise*(3), $a(X, U)$, $s(U, Y)$.

The hope is that *tortoise*(3) will fail often enough to break infinite loops.

However, certain not so pathological configurations will cause p not to terminate. Here is a simple example of one:

$a(1,2)$	$a(2,1)$
$a(1,3)$	$a(3,1)$
$a(2,3)$	$a(3,2)$

5. ADAPTING FOR TAIL-RECURSION OPTIMIZATION

Tail-recursion optimization (TRO) is a space saving technique that allows a subgoal to be removed from the stack while it is still open, under the right conditions. Briefly, the conditions are:

(1) it is the last subgoal on its "level";

(2) it has just undergone goal reduction using its last possible rule.

When these conditions hold, the subgoal is removed and the level above drops into its place. Because of the second condition, and the possibility that a new rule for the subgoal might be asserted on the fly in an interpreted environment, TRO is normally only applied to compiled procedures.

Example 5.1. Suppose the interpreter is processing the logic program $\mathbf{P}_5$ with rules

R1:	$p(X, Y) :- a(X, Y)$.	R4:	$a(1,2)$.
R2:	$p(X, Y) :- p(X, U), p(U, Y)$.	R5:	$a(2,1)$.
R3:	$r(X, Y) :- p(X, U), b(U, Y)$.	R6:	$a(2,3)$.
		R7:	$a(3,4)$.
		R8:	$b(4,5)$.

When given the top-level goal $r(X, 5)$, after some goal reductions and backtracks, the *goal stack* and *goal list* develop as shown on the left in Figure 6.

The stack goal $p(2, U_1)$ at level 4 has no goals to the right on its level. It has just been reduced using its last rule, R2, so no further backtracking of it is possible.

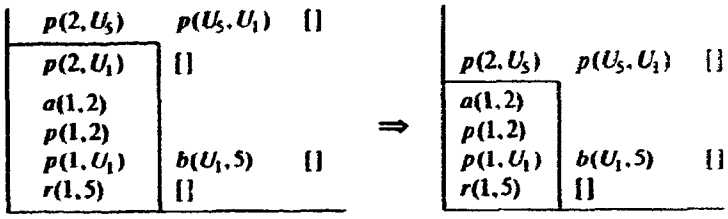

FIGURE 6. The TRO transformation.

Therefore TRO can "squash" it out of the stack as shown on the right, thereby reclaiming its space.

The problem of combining the tortoise-and-hare technique with TRO is obvious: the goal that demonstrates the loop might have been "squashed out" by TRO before the tortoise got to it. A major benefit of TRO is that it enables the same stack frame to be reused arbitrarily often during a recursive descent. On the other side of the coin, TRO makes it possible for PROLOG to go into a hard loop—one that does not even terminate by running out of stack space. Therefore, having a loop-detection mechanism in a TRO environment can pay even larger dividends.

We shall just sketch a solution to the problem of making the tortoise-and-hare compatible with TRO, and leave the messy details to interested implementers. The basic idea is for TRO to work by squashing three stack frames into two, instead of two into one. A cycle-length field L is maintained and initialized at 2. Steps that would add to the stack are counted through a cycle. *During* a cycle the top frame is squashed into the middle one, and the lower frame of the three is left intact. At the *end* of a cycle, the middle frame is squashed into the lower one, the top frame is squashed into the middle one, and the value of the cycle length L is incremented by 1. The tortoise always remains at least one level below the hare.

Thus if TRO has the recursion trapped in a set of three frames, eventually the tortoise will sit at the lower one and the hare at the middle one. (The upper one is essentially just a buffer.) But the "distance" in terms of goal reductions between them keeps cycling from 1 to L for increasingly large values of L. If the recursion is in a loop of length k, eventually $L \geq k$, and the loop is detected.

6. CONCLUSION

We have presented a method of recursive loop detection that can be implemented with low overhead in a conventional PROLOG interpreter. We have sketched how to make it compatible with TRO. We have argued that the main use of loop detection is as a debugging tool, somewhat analogous to subscript checking in other languages.

REFERENCES

1. Brough, D. R. and Walker, A., Some Practical Properties of Logic Programming Interpreters, in: *International Conference on Fifth Generation Computer Systems*, Institute for New Generation Computing, Tokyo, Japan, 1984, pp. 149–156.

2. Clark, Keith L. and Tärnlund, Sten-Åke, *Logic Programming*, Academic, London, 1982.
3. Knuth, Donald E., *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, 2nd ed., Addison-Wesley, Reading, Mass., 1981.
4. Kowalski, R. A., *Logic for Problem Solving*, North-Holland, Amsterdam, 1979.
5. Poole, David and Goebel, Randy, On Eliminating Loops in Prolog, *SIGPLAN Notices* 20(8):38–40 (1985).