Selecting Choice Points in An Intelligent Backtracking Schema

Ilyas Cicekli
Dept. of Comp. Eng. and Info. Sc.
Bilkent University
06533 Bilkent, Ankara, Turkey
e-mail: ilyas@bilkent.edu.tr

Abstract

We present a runtime intelligent backtracking method for prolog programs to avoid redundant failures. The method presented in this paper selects the best choice point as a backtracking point during the failure of a procedure call to avoid the same failure. The chosen backtracking is the best choice point that can be determined in runtime during execution of a goal without a further analysis. The modification of the Warren Abstract Machine [War83] is kept in minimum to implement our schema.

1 Introduction

The backtracking method used in a standard Prolog implementation is *naive* backtracking. In the *naive* backtracking, when a goal fails the backtracking is done to the most recent choice point (last alternative) although this choice point may be nothing to do with that failure. In this approach, a lot of unnecessary backtrackings will be done while the same failure occurs many times. An intelligent backtracking method analyze failures of a procedure call to choose the proper choice point to avoid redundant backtrackings. The chosen choice point may not be the most recent choice point. In other words, alternatives of choice points between the most recent one and the chosen one are discarded without retrying them. If they were retried, the system would have reencountered with the same failure.

Many intelligent backtracking schemes [Bru84, Cha85, Cod88, Cod91, Cox81, Lin87, Lin88, Per82] are presented to avoid unnecessary backtracking steps. Early works in intelligent backtracking [Bru84, Cox81, Per82] are implemented as prolog interpreters. Implementations of later works [Lin87, Lin88, Cod88, Cod91] are WAM based systems. Direct comparisons of early works with later works may not give fruitful results because the machinery used in early works are much slower than the WAM based implementations in later works. An implementation of an intelligent backtracking mechanism in a prolog interpreter may get much speed-up in terms of cpu time than an implementation of the same mechanism in a WAM based system. The reason is that the overhead of

the intelligent backtracking may be more costly in a WAM based system because a WAM based system is a very efficient implementation of Prolog.

An intelligent backtracking method can be only useful if our gains from it are much greater than its overheads. So, the chosen intelligent backtracking method should have small overheads. The overhead can be only minimum with small and localized extensions to the original system. For this reason, most of the intelligent backtracking methods mostly concentrate on unification algorithm which is the place of a failure and failure routine that performs backtracking operation.

All intelligent backtracking methods for Prolog assume that skipped alternatives are procedures without any side effects. For example, skipped alternatives wouldn't have done any *assert* or *retract* operations, or any input or output. If this is the case, a prolog system with an intelligent backtracking method may behave differently than a normal prolog system.

The intelligent backtracking schema presented in this paper is implemented as an extension of the WAM same as systems in [Lin87, Cod88]. Our mechanism is similar to mechanisms used in those systems except that how we keep reasons of a failure and how we find them. In the following sections, details of our system are given. In Section 2, a general discussion about failure and intelligent backtracking is given. After the implementation of our schema is given in Section 3, its performance results are discussed in Section 4.

2 Analysis of Failure

A failure normally occurs in a prolog system when the unification algorithm fails to match two terms. Since a failure occurs, the system should backtrack and try another alternative. The question is that how far the system should backtrack to find a new alternative. A normal prolog system naively tries the most recent choice point. But this most recent alternative may not fix the cause of the last failure and the same failure may occur again. We want to choose a proper choice point in an intelligent backtracking mechanism so that we will not encounter with the same failure.

When a procedure call fails, the system should backtrack a choice point between the most recent one and the shallow backtracking point. The shallow backtracking point is the choice point of the procedure being called or the choice point of its first ancestor which has a choice point. The shallow backtracking point indicates the furthest choice point to which the system can backtrack during a failure. If the shallow backtracking point is the most recent choice point, this is known as shallow backtracking. In that case, we have to try the most recent one. If the shallow backtracking point is not the most recent choice point, in that case the point that the system will backtrack depends on reasons of the failure of that procedure call. But, if reasons of the failure of that procedure call points to a choice point which is older than the shallow backtracking point, we can only again backtrack upto the shallow backtracking point. If they indicate a younger choice point, the system should backtrack to that choice point.

The procedure call s(X) fails when the goal p(X) is executed with respect to the prolog program in Figure 1.a. The reason of that failure is that the variable X was bound to 1 by the procedure q/1. In the time of that failure, we have three choice points. These choice points are choice points of procedures p/1, q/1 and r/1 in the creation order. The most recent choice point is the choice point of r/1. The shallow

```
p(X) :- q(X), r(X), s(X).            q(1).   r(1).   s(2).
p(X).                                q(2).   r(Z).
```

a. Program 1: Procedure Failure With Single Reason

```
p(X,Y) :- q(X), r(Y), t(Z), s(X,Y).   q(1).   r(a).   t(1).   s(1,b).
p(X,Y).                               q(2).   r(b).   t(2).   s(2,a).
```

b. Program 2: Procedure Failure With Two Reasons

Figure 1: Sample Prolog Programs For Failure

backtracking point is the choice point of p/1 since the procedure s/1 doesn't have a choice point and the procedure p/1 is the first ancestor of the procedure s/1 which has a choice point. So, when the procedure call s(X) fails we can backtrack upto the choice point of the procedure p/1 in the best case. The reason of the failure of the procedure call indicates the choice point of the procedure q/1 since that choice point was the most recent choice point when the variable X was bound to a value created by the first clause of q/1. Since the choice point of q/1 is younger than the choice point of p/1, we have to backtrack the the choice point of q/1. We don't have to retry alternatives of r/1 because they cannot fix the reason of that failure since a variable is bound to a value once in Prolog. If we submit another goal p(1) with respect to the same program in Figure 1.a, we will have a similar situation. The difference will be that the variable X is bound to a value created before the choice point of p/1. On other words, the reason of the failure points a choice point which is older than the choice point of p/1. The system should backtrack to the choice point of p/1 because that is the furthest choice point we can backtrack in that case.

In the previous example, the procedure call s(X) has just a single reason for the failure because the procedure s/1 has a single alternative. In general, a failure of a procedure call can have more than one reason. In this case, each reason will point a choice point and we choose the reason which points the youngest choice point as the reason of the failure of that procedure call. If we submit the goal p(X,Y) with respect to the program in Figure 1.b, the procedure call s(X,Y) will fail after two alternatives of the procedure s/2 are tried. In this case, we will have two reasons for the failure of that procedure call. The first one is that the second argument of the procedure call cannot unify with constant b and the second reason is that the first argument of the procedure call cannot unify with constant 2. The first reason points to the choice point of q/1 since the variable X is bound to a value created by the first clause of q/1. The second one points to the choice point of r/1 since Y is bound by the first clause of r/1. Since the choice point of r/1 is younger, it is chosen as a reason of the failure of that procedure call. In that time, the shallow backtracking point is the choice point of p/2 since s/2 doesn't have any choice point and p/2 is its first ancestor having a choice point. Since the choice point of p/2 is older, the choice point of r/2 is chosen as backtracking point for that failure.

3 Implementation of Intelligent Backtracking

To implement our intelligent backtracking method, we updated the Warren Abstract Machine (WAM) to incorporate our schema. In the rest of the discussion, a familiarity with the WAM is assumed. The full discussion of the WAM can be found in [War83] and [Aït91].

From the discussion in Section 2, our version of WAM has to have two main capabilities.

- It should be able to handle shallow backtracking points. At a certain time during the execution, the system should be able to access the current shallow backtracking point. It also should be able to maintain a shallow backtracking point link among choice points.
- It should be able to maintain reasons of the failure of a procedure call. It also should be able to find the choice point which was the shallow backtracking point during binding of the variable causing a failure.

In the following two subsections, we will discuss these two subjects. To get these capabilities, certain WAM instructions, the structure of choice points and the failure routine are changed in our implementation.

3.1 Shallow Backtracking Points

The shallow backtracking point at a certain time points to the choice point of the procedure that we are in or the choice point of one of its ancestors. If the procedure that we are in has a choice point, the shallow backtracking point will be its choice point. If it doesn't, the shallow backtracking point will be the choice point of its youngest ancestor of that procedure such that it has a choice point. An ancestor of a procedure is the procedure calling that procedure or ancestor of the calling procedure.

We will introduce a new register SB (Shallow Backtracking Point) to hold the shallow backtracking point. This register will point to a choice point in the local stack at a certain time. The register SB will be updated with a new choice point when that choice point created by a *try* instruction. The old value of the register SB will be saved in the new choice point by that *try* instruction. During a backtracking, the register will be set to point to the backtracked choice point. In fact, that choice point will be the most recent choice point at that time. Also, *trust* instructions will restore the register SB from the value stored in the choice point being discarded by that *trust* instruction.

This SB register is saved in environments by *allocate* instructions same as saving the environment register E except that the register SB is not updated by *allocate* instruction. The reason saving it in environments is that it can be restored by the value stored in the current environment by *proceed* instructions. When a *proceed* instruction is executed, we get out from a context of a procedure and return back to the context of one of its ancestors. Since that ancestor has an environment and its shallow backtracking point is saved in its environment, the register SB can be restored from that value.

In Figure 2, a sample prolog program and its local stack are given when the first clause of procedure t is entered. As seen in that figure, shallow backtracking points in choice points of procedures r and s point to the choice point of q. Since the register

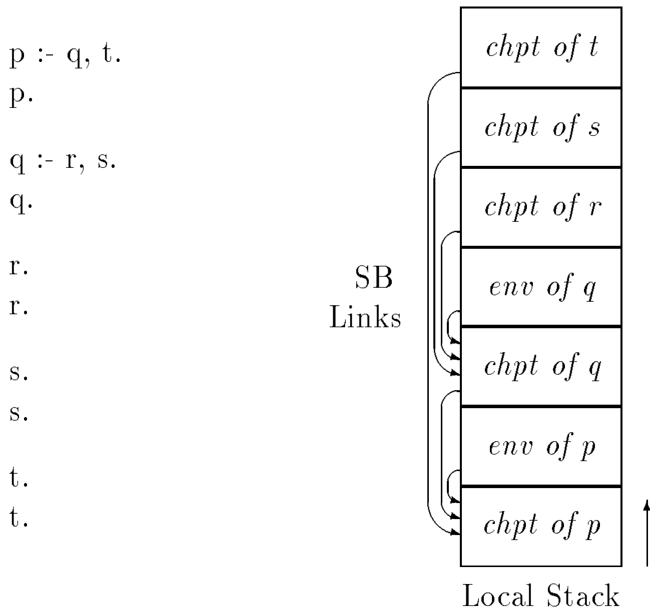

Figure 2: Shallow Backtracking Links

SB points to the choice point of procedure p when the first clause of p is entered, that value is saved in the environment of the first clause of p.

3.2 Finding Reasons of A Failure

A procedure call fails during unification of head arguments. Unification instructions used in head matching can be seen in two groups. Instructions in the first group (eg. *get_constant* and *unify_constant*) try to unify a variable with a specific ground term. A failure occurs when that variable is bound to a ground term which is different than that specific ground term. The second group instructions (eg. *get_val* and *unify_val*) are complete unification instructions which take two variables to be unified. In this case, a failure occurs when those variables are bound to two different ground terms. In the first case, the binding time of that variable determines the reason of that failure. In the second case, the youngest one of binding times of two variables plays role in the determination of the reason of that failure. In both cases, the reason of the failure will point to the choice point which was the shallow backtracking point during the binding causing that failure.

Since a failure of a procedure call can have more than one reason, our implementation should be able to handle these reasons. But we only need the youngest one of reasons of a procedure call failure because we choose the youngest one as a backtracking point. To store the youngest reason of a failure, we will reserve a space in choice points. After this point, we will call that field in choice points as RB (Reason Backtracking Point). This field will be initialized by a special value during the creation of that choice point by *try* instructions. During a failure, the RB field of the choice point indicated by the register SB may be updated with the reason of that failure. It is only updated with the reason of the failure, if that reason is younger than the value stored in RB field and older than the value in SB register. So, RB field will always held the youngest reason of a procedure failure. The RB field of a choice point may be copied into the RB field of its parent when that choice point is discarded. A parent of

a choice point is pointed by SB value saved in that choice point.

In this mechanism, we have to be able to find ages of variables. Since we are only interested in ages of variables causing failure, we have to only be able to figure out ages of variables bound to ground terms. The age of a variable depends on ages of trailed bindings in its dereference chain and age of the ground term at the end of its dereference chain. The age of a trailed binding is the choice point indicated by SB register during that binding. If there is no trailed bindings in the dereference chain of a variable, the age of that variable is same as the age of the ground term at the end of the dereference chain.

The age of a ground term can be found from its location in the heap. The age of a ground term is the choice point indicated by SB register during the creation of heap locations for that ground term. Because structures and lists are saved in the heap, their heap addresses help to find ages of variables bound to them. But variables bound to constants introduce some complications because constants may be copied directly into variables during bindings of those variables. So, we are not able to find ages of those kinds of variables in the original WAM. For this reason, constants are also put into the heap same as structures and lists, and variables are bound to these constants in the heap. On other words, constants are treated same as structures and lists in our mechanism.

The age of a trailed variable can be figured out from trailed bindings in the dereference chain of that variable. To find the age of a trailed binding easily, we use two new extra entries for each trailing operation. During the binding of a variable to a value, that variable is bound to the first extra trail entry and that trail entry is bound to that value. The value of SB register is stored in the second extra trail entry during that binding. From the stored SB value for a trailing operation, we find out the age of that trailed binding. The age of a trailed variable is equal to the age of the youngest trailed binding in its dereference chain.

4 Performance Results

We extended the byte emulator of the WAM based system of ALS (Applied Logic Systems) Prolog to implement our mechanism. To see gains and overheads of our system, we tested it with programs in three categories in addition to standard test programs for intelligent backtracking.

The first category includes programs which do a lot of unnecessary backtrackings in a regular prolog system. Of course, our system will give a good performance in these programs, because it will avoid a lot of redundant failures. A map coloring program given in Figure 3 can be a good example for this category. In a regular prolog system, there will be 147 failures before it gets a solution. In our mechanism there will be only 15 failures. There will be 5 times speed-up in our system in terms of cpu time for that example. In that program, the last subgoal fails for first values of variables B and C. In a regular prolog system, all alternatives of third and fourth subgoal will be tried before the second goal is retried to get another value for variable C although they are not responsible from the binding of variables B and C. In our schema, when the last subgoal is completely failed, the system will backtrack to the next alternative of the second subgoal without retrying third and fourth subgoals.

The second category contains prolog programs that do a lot of backtrackings but

```
mapcolor(A,B,C,D,E) :-
    next(A,B), next(A,C), next(A,D), next(A,E), next(B,C).

next(X,Y) :- next1(X,Y).
next(X,Y) :- next1(Y,X).

next1(green,red).
next1(green,yellow).
next1(green,blue).
next1(red,yellow).
next1(red,blue).
next1(yellow,blue).
```

Figure 3: Map Coloring

a few of them are redundant. This kind of programs will use all machinery in our schema without gaining anything. In fact, this kind of programs is the worst case for our schema. For example, if we put subgoal next(B,C) as third subgoal in the clause in Figure 3 instead of the last subgoal, there won't be any redundant failures. In that case, our system and a regular prolog system will do same backtrackings. Slow-down in our system will be 20 percent compared with the regular system.

Deterministic programs will be in the last category. Since most of overheads of our schema occurs during failure analysis, we want to see its effects on deterministic programs. We tested our system with a completely deterministic program, the slow-down in our system is only 1 percent. This result is a real encouragement because the overhead of our schema is minimum when there are a few failures. If there are a lot of failures, gains by avoiding redundant backtrackings will be more than overheads due to more complex failure routine.

5 Conclusion

An intelligent backtracking schema implemented as an extension of the WAM is presented in this paper. Extensions to the WAM are tried to be kept in minimum so that the overhead of the intelligent backtracking mechanism will be less. In the worst case, the overhead of our schema doesn't exceed 20 percent and it provides a good speed-up for non-deterministic programs. Our schema can be easily incorporated with a WAM based prolog compiler with a little bit of effort. This is thought as a future work.

Our mechanism chooses the best choice point for backtracking which can be determined in runtime without doing a further global analysis about the program. When we are designing our intelligent backtracking schema, we want to make sure that it can choose the best backtracking point which can be determined by a normal execution of a goal. Maybe more redundant choice points can be eliminated with a further global analysis, but this will bring more extra overhead.

References

[Aït91] Aït-Kaci, H., *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, Cambridge, 1991.

[Bru84] Bruynooghe M. and Pereiara L. M., *Deduction Revision by Intelligent Backtracking*, in Implementations of Prolog, ed. Cambell J. A., Ellis Horwood, 1984.

[Cha85] Chang J.-H. and Despain A. M., *Semi-Intelligent Backtracking of Prolog Based on a Static Data Dependency Analysis*, Proc. of 2nd Int. Symp. on Logic Programming, Boston, 1985.

[Cod88] Codognet P., Codognet C. and Filè G., *Yet Another Intelligent Backtracking Method*, Proc. of 5th Int. Conf. and Symp. on Logic Programming, Seattle, 1988.

[Cod91] Codognet P. and Sola T., *Extending the WAM for Intelligent Backtracking*, Proc. of 8th Int. Conf. on Logic Programming, Paris, 1991.

[Cox81] Cox P. and Pietrzylowski T., *Deduction Plans: A Basis for Intelligent Backtracking*, IEEE PAMI, Vol 3, 1981.

[Lin87] Lin Y-J. and Kumar V., *An Intelligent Backtracking Schema for Prolog*, Proc. of 4th Int. Symp. on Logic Programming, San Francisco, 1987.

[Lin88] Lin Y-J. and Kumar V., *A Data-Dependency Based Intelligent Backtracking Schema for Prolog*, J. Logic Programming, Vol 4, 1988.

[Per82] Pereiara L. M. and Porto A., *Selective Backtracking*, in Logic Programming, ed. Clark K. L. and Tarnlund S.-A., Academic Press, 1982.

[War83] Warren, D.H.D., *An Abstract Prolog Instruction Set*, SRI Technical Report 309, 1983.