IDNAF Prolog

S. D. GOODWIN
Department of Computer Science, University of Regina
Regina, Saskatchewan, Canada, S4S 0A2

M. MAHROOS AND E. NEUFELD
Department of Computational Science, University of Saskatchewan
Saskatoon, Saskatchewan, Canada, S7N 0WO

(Received January 1994; accepted February 1994)

Abstract—The relationship between resolution theorem proving and graph search is complicated by control statements that affect the program's meaning. An interesting case arises when combining Prolog interpreters implementing the cut and iterative deepening search. Because these features are dependent, an interpreter that naively combines both features gives unintended answers to queries. In particular, negation as failure breaks down.

One solution is to use sound negation, but many certain AI problems use both sound negation and negation as failure. As well, when debugging large Prolog programs, it is useful to be able to execute the same program with and without iterative deepening, even though the program contains cuts.

Besides generating unintended answers, the Prolog interpreter naively combining cuts and iterative deepening also generates redundant answers. Here, we show how to avoid both.

Keywords—Logic programming, Meta-interpreters, Iterative deepening.

1. INTRODUCTION

Although the resolution rule is complete for definite clauses, Prolog programs can generate infinite loops. A pathological example:

\[
\begin{align*}
  t(a, b). \\
  t(b, c). \\
  t(c, d). \\
  t(X, Z) :- t(X, Y), t(Y, Z).
\end{align*}
\]

In response to the query \( t(X, Y) \), this program loops infinitely before ever finding the answer \( t(b, d) \). The behaviour only worsens if we reorder clauses.

Although many interesting classes of loops can be detected [1], loop detection in general is undecidable. A breadth-first search (BFS) strategy instead of Prolog's depth-first search (DFS) finds all answers before looping, but it is well known that even for trivial programs the queue of partial proofs grows exponentially with the length of the derivation. Iterative deepening search (IDS) [2] combines the completeness of BFS with the space effectiveness of DFS (linear in the product of the branching factor and average proof depth). This technique—simply an itera-
tive bounded depth-first search, where the search is restarted with an increased bound on each iteration—requires no more space than DFS, and surprisingly, asymptotically no more time than DFS to find all answers. And like BFS, IDS is complete for pure Prolog.

It is natural to want to combine "impure" features such as negation as failure with IDS. During debugging, programmers can troubleshoot loops by turning on iterative deepening to see whether solutions exist at shallow levels. As well, many implementations of nonmonotonic reasoning formalisms use both classical negation (to represent negative knowledge) and negation as failure (to enable enable jumping to conclusions, "given no information to the contrary" [3]). One author found this combination useful for debugging the code generated by [4] that compiles default reasoning into Prolog. However, although Sterling et al. [5] discuss combining meta-interpreters that include bounded search, the cut is excluded. Stickel [6] describes an iterative deepening Prolog that uses a conditional cut to suppress redundant proofs of ground terms. In discussing enhancements, he describes similar problems that might arise trying to exploit intelligent backtracking techniques in the environment of iterative deepening.

Adding impure features to IDS introduces unintended answers, answers other than those that would have been generated if the program could terminate all infinite loops. To see why, note that negation as failure (+) is usually implemented as if written:

\[ +W \leftarrow \text{call}(X),!,\text{fail}. \]

where \text{call}/1 is a meta-predicate that attempts to find an answer to its argument when \( X \) is bound to a term, \text{cut} (!) commits Prolog to all choices made since the parent goal unified with the head of the current clause, and \text{fail} always fails. Clearly, an IDS Prolog, beginning with depth 0 and incrementing by 1 always returns the unintended answer yes to any proof of negation, which will affect other parts of the program. The problem is the meaning of the cut given the IDS search strategy and arises because we are composing meta-interpreters that implement dependent features: IDS guarantees search is exhaustive whereas the cut guarantees it is not.

The IDNAF (iterative deepening negation as failure) meta-interpreter solves the problem by distingushing depth failure from finite failure and using cut lookahead to avoid unintended branches given depth failure. This is inexpensive, consistent with programmers' intuitions, and has a meta-circular implementation. Thus, IDNAF can run other Prolog meta-interpreters implementing other pruning strategies such as linear resolution on ordered clauses [7].

On the \( m \)th iteration, IDNAF prints all answers found at depth \( m \) or lower. IDNAF2 eliminates this repetitive behaviour, and prints answers exactly once, without storing answers.

2. IMPLEMENTATION

To simplify, we don't give meta-circular interpreters [8]. The vanilla meta-interpreter is

\[ \text{pr}(\text{true}) \].
\[ \text{pr}((X,Y)) \leftarrow \text{pr}(X),\text{pr}(Y). \]
\[ \text{pr}(X) \leftarrow \text{clause}(X,Y),\text{pr}(Y). \]

where \text{clause}/2 is true if \( X :- Y \) is a program clause and the unit clause \( X \) is stored as \( X :- \text{true} \). The cut may be implemented as follows:

\[ \text{pr}(\text{true},...) \leftarrow !. \]
\[ \text{pr}(!,F) \leftarrow !,(\text{true},F=\text{backtracking through cut}). \]
\[ \text{pr}((X,Y),F) \leftarrow \]
\[ \text{pr}(X,F), \]
\[ (F=}=\text{backtracking through cut};\text{pr}(Y,F)). \]
\[ \text{pr}(X,_) \leftarrow \]
\[ \text{clause}(X,Y), \]
\[ \text{pr}(Y,F), \]
\[ (F=}=\text{backtracking through cut},!,\text{fail};\text{true}). \]
When the interpreter backtracks through a cut, F is instantiated and forces failure. (This interpreter results in unneeded choice points for "obviously determinate" programs that contain no cuts; see [8] for a discussion.)

Stickel [6] implements iterative deepening by bounding the number of subgoals proved. IDNAF bounds the height of the proof tree. The meta-predicate pr(Goal, DI, DF, Max) finds a proof of Goal, if one exists, from initial depth DI and final depth DF that cannot exceed Max:

\[
\text{pr}(\ldots, D, M) \leftarrow D > M, !, \text{fail}. \\
\text{pr}(\text{true}, D, D, \ldots) \leftarrow !. \\
\text{pr}(X, Y, D0, D3, M) \leftarrow !, \\
\text{pr}(X, D0, D1, M), \\
\text{pr}(Y, D0, D2, M), \\
\text{max}(D1, D2, D3). \\
\text{pr}(X, D1, D3, M) \leftarrow \\
\text{clause}(X, B), \\
D2 \leftarrow D1 + 1, \\
\text{pr}(B, D2, D3, M).
\]

For each value of M, pr/4 finds only those goals solved requiring exactly depth M.

The unintended answers of Section 1 result if one naively combines cuts with IDS by adding to pr/4 an extra parameter that detects backtracking through a cut (equivalently—by composing interpreters). To avoid this, clause 3 of pr/4 must distinguish three possibilities after proving X. One, finite failure before depth M. Two, success before depth M, and failure to succeed before depth M. In the first case, the meta-interpreter fails, and the caller makes another choice. In the second case, the IDNAF proof of Y proceeds as before. In the third case, IDNAF must not choose a branch that might have been pruned had X succeeded and a cut subsequently encountered. IDNAF solves this by using look_ahead(Y) to check for the presence of a cut in Y and executing it. In the third case, the IDNAF proof of Y proceeds as before. To do this, IDNAF adds a fifth parameter to pr/4 to distinguish the second and third cases. Figure 1 gives details.

\[
\text{pr}(\ldots, D, D, M, \ldots) \leftarrow D > M, !. \\
\text{pr}(!, D, D, \ldots, B) \leftarrow !, (\text{true;} B=\text{backtracking}). \\
\text{pr}(\text{true}, D, D, \ldots) \leftarrow !. \\
\text{pr}(X, Y, D0, D3, M, B) \leftarrow !, \\
\text{pr}(X, D0, D1, M, B), \\
(D1 = \leq M \rightarrow \\
(B = \text{backtracking} \rightarrow \\
D3 \leftarrow D1; \\
\text{pr}(Y, D0, D2, M, B), \\
\text{max}(D1, D2, D3)); \\
(D3 = D1, \\
(\text{look_ahead}(Y) \rightarrow \\
B=\text{backtracking}; \\
\text{true})));). \\
\text{pr}(X, D1, D3, M, \ldots) \leftarrow \\
\text{clause}(X, \text{Body}), \\
D2 \leftarrow D1 + 1, \\
\text{pr}(\text{Body}, D2, D3, M, B), \\
(B = \text{backtracking} \rightarrow (!, \text{fail}; \text{true})).
\]

Figure 1. A partial IDNAF interpreter.
A new problem: IDNAF reasons about depth failure, but not about finite failure, failing itself when failure occurs in the object program. Thus, IDNAF cannot distinguish proofs that occur at depth $n$ for the first time from proofs that occur at depth $n$ after a clause containing a cut failed at depth $n + m$. For completeness, all answers occurring at any level below $n + m$ must be printed. To eliminate redundancy and print each answer exactly once, it is necessary to reason about finite failure. IDNAF2 does this by adding a parameter to pr/5 to indicate finite failure. That is, a failed proof of Body in the last clause of Figure 1 must be made to "succeed" and signal finite failure at depth $D3$. Since IDNAF2 can detect finite failure that did not backtrack through a cut in a clause containing a cut, it can set the starting depth of the proof of the next clause selected to $D3$.

Having to reason about properties of previously failed OR-branches in the proof tree complicates the IDNAF2 meta-interpreter. Either failure driven clauses (e.g., the last clause of pr/5) must be rewritten iteratively or failure depths must be tracked by simulating global variables with assert and retract.

"Correctness" of IDNAF

Because cuts generally change the meaning of a program, we do not argue correctness in the sense of soundness and completeness, but as preserving the programmer's intended meaning.

Clearly IDNAF generates only intended answers. Recall the definition of SLD-tree [9] for a definite program $P$ and a definite goal $G0$ where the computation rule $R$ selects the leftmost literal of the goal clause. Then, the SLD-tree for $P,R$ is a possibly infinite labeled tree satisfying

1. $G0$ is the root of the tree
2. $Gi+1$ is a child of $Gi$ iff there is a renamed version of a clause $Ci$ in $P$ such that $Gi+1$ is derived from $Gi$ and the renamed $Ci$ by $R$. The edge connecting $Ci$ and $Gi+1$ is labeled with $Ci$.

If the programmer's intended meaning is the set of successful answers found at the leaf nodes by an inorder traversal of this tree, then an infinite branch blocks answers to its right.

Thus, specialize the definition of SLD-tree to SLD($m$)-tree, the subtree of the SLD-tree beginning at the root, and containing at most the first $m$ edges of any path to a leaf. Infinite branches no longer block answers to the right and IDNAF cautiously performs any pruning that might possibly occur along a infinite branch so no unintended answers result.

Furthermore, IDNAF2 prints each answer exactly once, at the "depth" it occurs. Allow the SLD($m$) tree to have branches of greater than unit length. Specifically, let the edge following a clause body containing a cut have length equal to the depth of the previous finitely failed derivation. This prevents a shallow solution from being "buried" by a cut. Also, the meaning of the tree is not changed by this "branch extension" and no finite derivation can become infinite.

REFERENCES