Automata-Driven Indexing of Prolog Clauses †

R. Ramesh
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083.

I.V. Ramakrishnan and D.S. Warren
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794.

Abstract

Indexing Prolog clauses is an important optimization step that reduces the number of clauses on which unification will be performed and can avoid the pushing of a choice point. It is quite desirable to increase the number of functors used in indexing as this can considerably reduce the size of the filtered set. However this can cause an enormous increase in running time if indexing is done naively. This paper describes a new technique for indexing that utilizes all the functors in a clause-head. More importantly, in spite of using all the functors, this technique is still able to quickly select relevant clause-heads at run time. This is made possible primarily by a finite-state automaton that guides the indexing process. The automaton is constructed at compile time by preprocessing all the clause-heads.

1 Introduction

The fundamental computational step in execution of Prolog programs is the selection and unification of clause-heads with goal. A successful unification results in the creation of several subgoals and each of these in turn must be unified with additional clause-heads. This process continues until either all the subgoals created are satisfied by the facts or one of them fails to unify with any clause-head. Thus the repeated selection of unifiable clause-heads is an important operation critical to the efficiency of Prolog program execution. Developing techniques to significantly enhance the speed of this selection process is therefore a problem of practical importance to Prolog compilation and execution technology.

†Research partially supported by NSF grant CCR-8805734

Fast selection techniques that have been proposed, typically first filter the clause-heads to form a (presumably small) set that are likely to unify and then perform unification on each of the filtered clause-head in this set. These techniques can be broadly grouped into two classes.

The techniques in the first group basically index on the outermost functor of one or more arguments in the clause-head. A hash table is built on these functors which is then accessed for retrieving the filtered set of clause-heads. This approach is quite popular as seen by its use in the WAM (warren abstract machine)[12], Quintus Prolog [9], Stony Brook Prolog [11] and several other Prolog systems [2,13]. The problems with this approach are that firstly it fails to distinguish between distinct clause-heads that do not differ in the functor of the argument indexed on. For instance, by indexing on the outermost functor of the first argument it fails to distinguish $p(f(a,b),c)$ from $p(f(a,c),d)$. Secondly, if the goal has a variable corresponding to the argument indexed on then again it will fail to distinguish between clause-heads. These two situations can sometimes be handled by allowing the programmer to specify the indexing argument. The selection process now is no longer transparent and to write efficient programs the programmer must be aware of the indexing method used in the Prolog system and organize the program to exploit it effectively.

The second group of techniques transforms every clause into a binary codeword by first transforming each of its argument into a codeword and then OR'ing all of them together. The filtered set is obtained by searching for the goal's codeword among the codewords for clause-heads. This technique has been studied in [10]. The problem with this method is that such an encoding is an imperfect representation of a clause-head. Specifically, important structural information in clause-heads is lost in the transformation process. Thus, although $p(a,b)$ and $p(b,a)$ are structurally dissimilar, yet they are assigned the same code. Note that this method is

well-suited for database applications where arguments are atomic. However, Prolog clauses are complex structures containing variables. Known transformation techniques for dealing with them are quite ad hoc and result in significant loss of structural information quite critical for filtering.

Increasing the number of symbols used in indexing can result in reducing the size of the filtered set. But doing so can increase running time if indexing is done naively. A 'good' indexing technique therefore should be able to utilize all the (non-variable) symbols in clause-heads without losing significant structural information and have good run time performance. In this paper we address the important problem of designing such a technique.

1.1 Brief Overview and Related Results

We describe a new technique for indexing Prolog clause-heads. Unlike known techniques it utilizes all the non variable symbols in clause-heads for indexing. Furthermore even structurally different clause-heads containing the same functors (such as $p(a,b)$ and $p(b,a)$) are distinguished. In our approach each clause-head is transformed into a set of strings by doing a left-to-right preorder traversal and removing nodes labelled with variables. Thus $f(a,g(X,b))$ is transformed into fag and b. Observe that the *clause-head strings* so obtained contain all the non variable symbols in its head. As each function symbol is assumed to have unique arity in Prolog implementations, these strings also retain the head's structure, i.e., we can reconstruct the clause-head given its preorder sequence.

Given a goal we perform a series of complex string matching steps involving clause-head strings and goal strings. The outcome of these steps are then correlated to obtain the filtered set. These string matching steps require repeated inspection of goal strings and therefore performing each such step independently is inefficient. Hence we construct a finite-state automaton (based on the clause-head strings) and use it to guide our indexing process. The information embodied in the states of the automaton is now used to avoid reinspection of sequences of symbols in goal strings and thereby improve the performance of our technique. The following are some important features of our technique.

- The finite-state automaton is constructed at compile time by preprocessing all the clause-heads.
- Selecting a clause-head at run time is accomplished within time (in the worst case) proportional to the number of variables in the clause-head and goal.
- Time to construct the automaton (at compile time) and its space requirements are both quadratic in the size of the clause-heads (in the worst case). Both can be made linear by increasing the constant in the running time of clause selection and we briefly outline how this is accomplished.
- An important aspect of our technique is that a selected clause-head will unify with the goal if they are both *linear* (i.e., each variable is restricted to at most one occurrence). In other words among the selected clause-heads only those that have repeated variables may fail to unify. Often clause selection in typical Prolog programs involves linear terms only. This fact can be gainfully exploited by our technique to obtain the unifier during the indexing process. In contrast, known methods do this by performing unification on each clause-head in their filtered set (which is at least as large as the one constructed by our technique). Unification of each such clause-head requires time proportional to the sum of its size and that of the goal (using known linear-time unification algorithms in [8]).
- Our technique generalizes the known methods of indexing on functors of specified arguments. It is also transparent to the programmer who need no longer organize the program to effectively exploit the indexing method used by the compiler.

Finally, we also discuss another indexing technique (that also utilizes all the functors in a clause-head) for a machine model in which we can perform bit-string operations of union and intersection in constant time. This technique is of practical importance as long as the number of clause-heads with the same predicate name does not exceed the wordsize.

This paper is organized as follows. In the following section issues related to compiling clause-heads for fast indexing are identified. Based on these issues we describe their compilation into a finite-state matching automaton in section 3. The clause-heads are selected based on the outcome of scanning the goal with the automaton at run time. A method to do this scan efficiently is outlined in section 4. A variant of the indexing technique that results in reducing the space requirements of the automaton is described in section 5. In section 6 we describe another indexing technique using bit string operations. Concluding remarks appear in section 7.

1.2 Notations

A *term* is either a variable or an expression of the form $f(t_1,t_2,\ldots,t_n)$ where f is a functor of arity $n \geq 0$ and $t_1,t_2,\ldots,t_n$ in turn are also terms.

We adopt the standard Prolog convention of using capital letter for the first symbol in a variable name. Thus $f(a,g(X,b))$ is a term with X as the variable, f and g

```
begin
fail:=false;
repeat
   If G(p_g) and R(p_r) are both functors
      then begin
         If G(p_g)≠ R(p_r) then fail=true
         else begin p_g:=p_g + 1; p_r:=p_r + 1 end
      end else
         if one of them is a variable, say G(p_g), then
         begin p_g:=p_g + 1;
            advance p_r to node immediately
            following the subtree rooted at R(p_r)
         end
until (fail=false) or
      (G and R are completely scanned)
end.
```

Fig. 1: Selection Algorithm

as functors of arity 2 and a and b as functors of arity 0.
A *term tree* is the standard tree representation of a term (see fig.2).
A term is *linear* if each variable in it occurs only once.

2 Issues in Selecting Clause-Heads

We first identify issues involved in indexing clause-heads (called rules from now on) through a simple selection algorithm.

2.1 Simple Selection Algorithm

Our selection algorithm uses all those nodes in the goal and rule that do not occur within variable substitutions when the two are unified. Specifically, it selects a rule if and only if it unifies with the goal after uniquely renaming multiple occurrences of variables in both the rule and goal. (Note that such renaming makes them both linear terms.)

The structure of rules selected by our selection algorithm is described by the following intuitive picture. First superpose the goal and rule trees at their roots. Next mark all those nodes that fall on variables. The rule is selected if and only if after deleting all such marked nodes and their subtrees, the two trees are isomorphic.

Fig.1 is an outline of such a selection algorithm. The rule and goal trees are traversed in preorder and stored in arrays R and G respectively. Two pointers, p_r and p_g, are used to scan R and G respectively. The rule is selected if upon termination *fail* is false. The trouble with this selection algorithm is that its running time is proportional to the number of nodes in the goal and rule trees (in the worst case). Note that within this time bound we can in fact unify them using well known linear-time unification algorithms (such as [8,6]).

We now examine issues related to improving considerably the running time of our simple selection algorithm.

2.2 Improving Running Time

Observe that our simple selection algorithm cycles between two phases - *match* and *skip*. In each step the phase is first determined and then the computation appropriate to that phase is performed. Transition between phases occurs as follows. If the algorithm is in match phase and the nodes currently being compared are both functors then it continues to remain in the same match phase. On the other hand a new match phase is entered if it is currently in a skip phase and the nodes being compared are again functors. Finally, it enters a new skip phase whenever one of the nodes being compared is a variable. The computations performed in the two phases are as follows. If the pair of functor symbols compared in a match phase are identical then p_g and p_r are both incremented by one. A mismatch on the other hand, results in the rule being discarded. For the skip phase, suppose (without loss of generality) p_g points to a node labelled with a variable, say X, and p_r to some node, say v. Then p_g is advanced by one whereas p_r skips the entire subtree rooted at v and advances to the node immediately following the last node in the skipped subtree in preorder. We say that the subtree at v is the *substitution* computed for X.

Note that the total number of distinct phases the algorithm goes through is proportional to the *number of substitutions computed* which in the worst case is at most equal to the total number of variables in both the goal and rule. Also note that each skip phase can be accomplished in $O(1)$ time by keeping a pointer with every node v (in arrays G and R) to the node that appears last in the preorder traversal of the subtree rooted at v. Observe that if we can accomplish each match phase also in $O(1)$ time then the worst case running time of our selection algorithm is proportional to the number of substitutions computed. We now examine issues related to doing the match phase in $O(1)$ time.

2.3 String Matching Operations

The computation performed in a match phase is basically comparing pairs of functor symbols in succession. If we can compare this entire sequence of functor pairs in $O(1)$ time then the match phase requires only

$O(1)$ time. Towards this objective we examine below the kinds of string matching questions that can possibly arise in a match phase. We will denote the string of functors separating two consecutive variables in the goal's (rule's) preorder as *goal (rule) strings* (see fig.2 below).

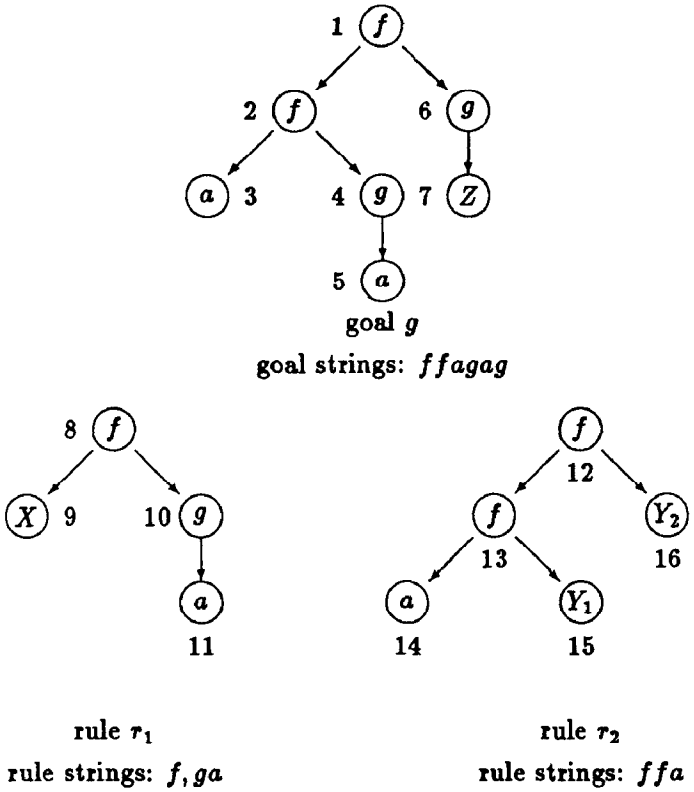


Fig. 2

Observe that the two variables, for which substitutions are computed in the skip phases immediately preceding and succeeding a match pahse, are either both rule variables (such X_i, X_{i+1} in fig.3(a)) or are both goal variables (such as Y_j, Y_{j+1} in fig.3(c)) or one is a goal variable and the other a rule variable (such as X_i, Y_{j+1} in fig.3(b) and Y_j, X_{i+1} in fig.3(d)). In fig.3 α and β denote rule and goal strings respectively and the preorder traversals of the rule and goal are stored in arrays R and G respectively. The pair of arrows leaving a variable mark the two ends (in R or G as appropriate) of the subtree computed as its substitution (such as a and b for X_i in fig.3(a)).

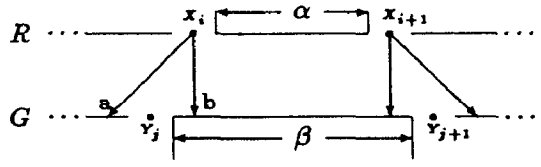


Fig. 3a

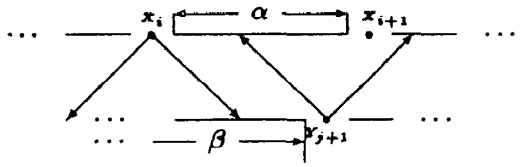


Fig. 3b

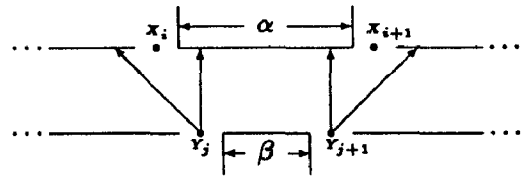


Fig. 3c

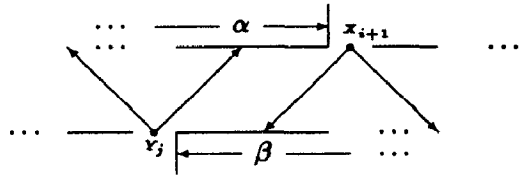


Fig. 3d

Each of these four situations gives rise to a different string matching question in the match phase as follows.

1. Does α (rule string) occur in β (goal string)? (Fig.3(a)).
2. Does a prefix of α match a suffix of β? (Fig.3(b)).
3. Does β (goal string) occur in α (rule string)? (Fig.3(c)).
4. Does a prefix of β match a suffix of α? (Fig.3(d)).

Observe that these four questions are a special case of the generic question - Does the prefix of a rule (goal) string occur in a goal (rule) string? We now show how to compile the rule strings into a finite-state automaton that at run time will enable us to answer these questions in $O(1)$ time.

3 Compiling Automata

Central to our technique is a finite-state automaton constructed from the rule strings. Such an automaton was first conceived by Knuth, Morris and Pratt [5] for recognizing instances of a single keyword string in a text string. Aho and Corasick [1] extended it to handle multiple keyword strings. This automaton is a trie in which each node denotes a state. From each state the automaton either makes a *goto* transition or a *failure* transition depending on whether the next symbol in the input extends the keyword prefix matched so far. Fig.4(a) is such an automaton for the three rule strings generated from the rules in fig.2

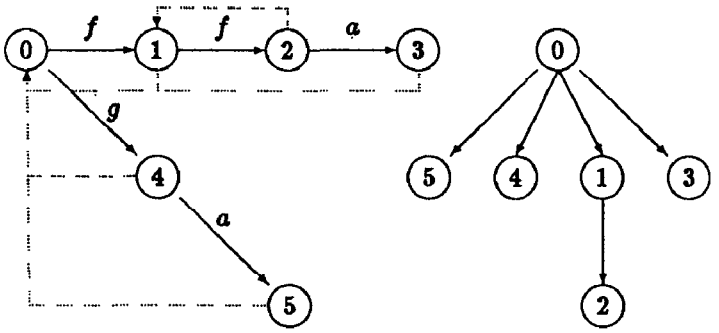


Fig. 4a: Automaton

Fig. 4b: Fail Tree

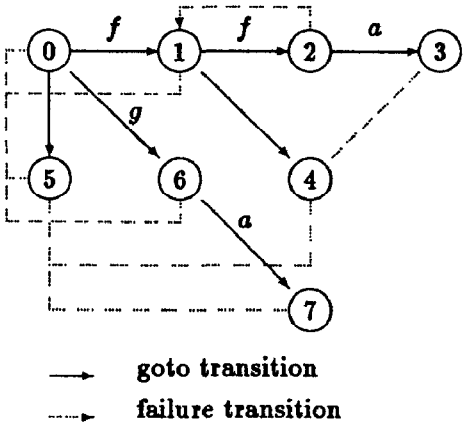


Fig. 4c: Automaton

Time required to construct this automaton is linear in the size of the keywords and if the alphabet set is fixed then the space required by it is also linear in the size of the keywords [1]. The following two properties of the automaton form the basis of our compilation.

1. Every prefix of a keyword string is represented by a state in the automaton.
2. If the *failstate* of i is j (i.e., the failure transition from i goes to j) then j represents the longest proper suffix of the string represented by i. (We say state i *represents* string σ if the path in the automaton from its start node to i spells σ. Thus state 2 in fig.4(a) represents ff.)

The main problem with this automaton is that (as is) it is only able to tell whether an entire keyword string occurred in the input text. This is all that is needed for rule selection when the goal is *ground* as in functional/equational programming. However recall that in the presence of variables in the goal we need to know whether a prefix of a rule (goal) string occurs in a goal (rule) string.

We first extend the automaton to handle such questions. The rule strings generated from the clause-heads of the Prolog program form the keywords of this automaton. At run time the automaton scans the symbols of the goal tree in preorder. Suppose we want to know whether the prefix α of a rule string, say r_i, matches the substring β of goal (see fig.5(a)).

Let s_β denote the state of the automaton after reading the last symbol in β. Let s_α be the state representing α. (Recall property (1).) Then,

Theorem 1 *α matches β iff s_α is reachable from s_β through zero or more failure transitions only.*

The proof of the theorem follows from property (1). Details are omitted.

Observe that each state has a unique fail state. So by deleting all the goto transitions and reversing the directions on failure transitions we obtain the *fail tree* of the automaton. (Fig.4(b) is the fail tree for the automaton in fig.4(a).) To each node in this fail tree we assign its preorder number (pre) and the number of descendants (nd) in its subtree. Note that all this preprocessing is done at compile time. For α to match β, s_α must be an ancestor of s_β in the fail tree (i.e., verify $pre(s_\alpha) \leq pre(s_\beta) \leq pre(s_\alpha) + nd(s_\alpha)$). Since this can be verified in $O(1)$ time we can therefore answer in $O(1)$ time whether a prefix of a rule string occurs in a goal string.

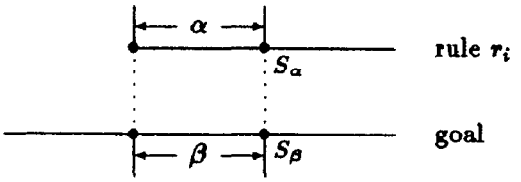


Fig. 5a

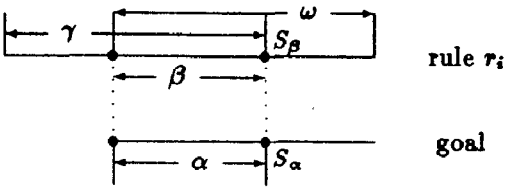


Fig. 5b

Now we extend the automaton to answer whether a prefix α of goal string matches a substring β of a rule string (see fig.5(b)). s_α is the state of the automaton on scanning α and s_β is the state corresponding to prefix γ of rule string. Note that the goal strings are not available at compile time. As the automaton is constructed without these strings we can no longer guarantee that every prefix of a goal string will be represented by a state in the automaton. Hence theorem 1 is no longer useful to answer questions related to prefixes of goal strings. Specifically, even if α does not match β, s_α can still be an ancestor of s_β in the fail tree. For instance,

the automaton in fig.4(a) ends up in state 1 upon scanning the prefix gf of a goal string. Now observe that even though 1 is an ancestor of 2 in the fail tree, gf does not occur in the rule string ffa. The problem is that s_α is an ancestor of s_β even if a suffix of α occurs in β (such as f in the above example). To overcome this we must ensure that if a prefix of a goal string matches a substring of a rule string then that substring is a prefix of some keyword in the automaton. (Obviously if the automaton was constructed using the goal strings also then this is easily ensured.) However note that we do not need to represent every prefix of a goal string in the automaton. We need only those that match substrings of rule strings. Based on this important observation we now show how this can be accomplished using rule strings only!

In fig.5(b) suppose α matches β. Now observe that β is a prefix of ω which in turn is a suffix of a rule string. Thus all we need to do now is to make every suffix of a rule string into a keyword of the automaton. (Thus, in addition to the rule strings in fig.2 we now insert their suffixes a, fa also into the automaton, in fig.4(a), as its keywords. Fig.4(c) is the resulting automaton.) Finally, note that even after doing so s_α will be an ancestor of s_β if any suffix of α occurs in β. We can eliminate such possibilities by

Theorem 2 *α matches β iff s_α is an ancestor of s_β and depth(s_α) in the goto tree*[1] $= \mid \alpha \mid$.

The theorem's condition is again verifiable in $O(1)$ time. Thus all string matching questions can be answered in $O(1)$ time.

Finally note that the size of the all the keywords inserted into the automaton (rule strings and all their suffixes) can now become quadratic in the size of the rule strings (in the worst case).

4 Scanning Goal

To begin our selection we have to first scan the goal tree in preorder with the automaton and store its state transitions in an array. Clearly, for efficiency reasons we should not inspect regions of goal that occur within a variable substitution of every rule. For example, in fig. 2 the goal's subtree $g(a)$ occurs inside substitutions for X (in rule 1) as well as Y_1(in rule 2).

We now outline the essential ingredients involved in avoiding inspection of such subtrees during selection. The main idea is to inspect the goal on demand only to the extent needed to perform a string match. Rules are in two states - *active* and *suspended*. Initially all

[1] The *goto tree* of the automaton is obtained by removing all its failure transitions.

location	1
node labels	f
state	1

Fig. 6a

location	1	2	3
node labels	f	f	a
state	1	2	3

Fig. 6b

location	1	2	3	4
node labels	f	f	a	?
state	1	2	3	?

Fig. 6c

location	1	2	3	4	5
node labels	f	f	a	?	g
state	1	2	3	?	6

Fig. 6d

of them are active. To begin with we pick any active rule and inspect the first few functors of the goal (in preorder) that are necessary to perform the first string match involving the selected rule. Suppose the skip phase immediately following the match phase is *triggered* by a rule variable[2]. We then suspend this rule on that node in the goal tree that immediately follows the skipped subtree in preorder. In general a rule, say r_i, is suspended only when the skip phase is triggered by r_i's variable and there are nodes in the skipped subtree that have not been inspected. A suspended rule is reactivated whenever the node, on which it is suspended, is either inspected in order to perform a string match step on behalf of an active rule or it becomes the root of a substitution. In case all rules become suspended then we reactivate the rule that is suspended on the node farthest from the root of the goal tree. The algorithmic details are a bit involved and are omitted. We illustrate this scanning process on the goal and rules in fig.2 using the automaton in fig.4(c).

Initially, r_1 and r_2 are active. States 1 and 7 represent the first and second strings of r_1 respectively (f and ga) and state 3 represents the only string (ffa) of r_2. To begin with we can choose any rule, say r_1. Before we can match its first string we have to scan the goal. As the length of this string is 1, we inspect only one goal symbol which in this case is its first symbol. (The inspected symbols of the goal are stored in an array.)

The first string match step of r_1 succeeds as the state representing its first string and the state of the automaton on reading the first goal symbol are the same (fig.6(a) shows the contents of the goal array on inspecting its first symbol). The next phase is a skip (triggered by X). The subtree rooted at 2 in the goal is skipped

[2] A skip phase is said to be triggered by a rule (goal) variable if in this phase a substitution is computed for the rule (goal) variable.

and r_1 is suspended on node 6 in the goal tree. Following this we choose the next active rule (r_2) and perform its first string matching step. To do this we need to scan the goal further as the symbols inspected so far (which is 1) is shorter than that required to perform this step. We scan two more symbols (see fig.6(b)).

The first string match of r_2 succeeds as the state representing its first string ffa is the same as that in the 3rd entry of the goal array. In the following skip phase, triggered by Y_1, the subtree at 4 is skipped. Following this r_2 is also suspended at node 6. Now note that both rules are suspended. This means that the subtree at 4 occurs inside variable substitutions of both rules. Therefore we need not inspect any symbols within this subtree. In the array this is represented by a dummy variable (see fig.6(c)).

At this point both are suspended on the same node. So we can activate any one, say r_2. (In general we choose the rule suspended on a node farthest from root.) For r_2 the step to be performed now is compute Y_2's substitution. In this skip phase the subtree at 6 is skipped. r_1 now is reactivated as the node on which it was suspended (node 6) has become the root of a substitution. Following this skip phase r_2 is selected as there is no node in the goal tree that follows the subtree at 6 in preorder.

Since r_1 is the only active rule left we begin its second string match. To do this we have to inspect the symbols in nodes 6 and 7 (see fig.6(d)). Since 7 is labelled with the variable Y we therefore stop the scan and perform the match. Notice that for inspecting these two nodes state 0 was used as initial state of the automaton. This is because the symbols inspected in this scan constitute a different goal string. The preceding goal string terminated before the dummy variable. Observe that the second string matching step of r_1 involves matching the prefix g of rule string ga with the goal string g. This also is successful as the state of the automaton on inspecting node 6 of the goal (state 6) also represents the prefix g of the rule string ga. In the following skip phase, triggered by Z (goal variable), the subtree rooted at node 11 in r_1 is skipped. As this is the last node in r_1's preorder it also is selected.

4.1 Coarse Filtering

The selection algorithm (described so far) regards every rule as a likely candidate for inclusion in the filtered set and so processes each of them separately. Thus it will unnecessarily examine a rule even if the functor symbol at its root differs from that of the goal's root. In contrast note that an indexing technique that is based on hashing the root functor symbol, will not even examine such rules.

We now modify the selection algorithm to avoid such needless computations. Specifically, we first construct a coarsely filtered set of rules such that the first string of every rule in this set is either a prefix of the goal's first string or vice versa. (Note that the first string in every rule always begins with the root's functor symbol.) More importantly, rules that do not belong to this set are not looked at during its construction.

Let g_1 denote the goal's first string. Let $S_1 = \{r \mid$ the first string of r is a prefix of $g_1\}$ and $S_2 = \{r \mid g_1$ is a prefix of the first string of $r\}$. The following is a description of the ideas underlying the construction of the coarsely filtered set $S_1 \bigcup S_2$. But first we need the following concept. We say that A is the *primary accepting* state of a keyword string α if it is both an accepting state for α and represents α. For instance in fig. 4(a), both 1 and 2 are the accepting states of the rule string f. But only state 1 is its primary accepting state.

Suppose $r \in S_1$ and its first string β matches a prefix α of g_1. Since β is a keyword string of the automaton, one of its accepting states is a primary accepting state. Now note that the path from the start state to this primary accepting state spells out β. This implies that α can be entirely scanned by the automaton without making any failure transitions. Based on this observation S_1 can be constructed as follows. The automaton scans the symbols in g_1 (from left to right) and makes transitions. It continues scanning these symbols as long as it makes only goto transitions. During such a scan if the automaton makes a goto transition to the primary accepting state of β then rule r is included in S_1.

For constructing S_2, suppose $r \in S_2$ and g_1 is a prefix of its first string β. Once again the automaton can scan g_1 entirely without making any failure transitions. Let A denote the state of the automaton on completely scanning g_1 without making any failure transitions. If g_1 is a prefix of β then the primary accepting state of β must be a descendant of A in the goto tree. Therefore S_2 will consist of only those rules such that the primary accepting states of their first strings are descendants of A in the goto tree.

During compilation we maintain the following information. With each state A, we keep a set C_A of all those rules for which A is the primary accepting state of their first strings. (Note that all rules in C_A should have identical first strings.) We also maintain another set D_A of rules such that the primary accepting states of their first strings are descendants of A in the goto tree.

At run time the automaton starts off by reading the symbols in g_1. It continues scanning them as long as it makes only goto transitions. During this scan if it enters an accepting state A then the rules in C_A are

added to S_1. The scanning process is suspended when either g_1 is completely scanned without making any failure transitions or a failure transition occurs before all the symbols in g_1 have been read. In the former case, if B is the state of the automaton on completely scanning g_1 then $S_2 = D_B$ whereas in the latter case $S_2 = \emptyset$. In either case construction of the coarsely filtered set is now complete. We resume the scan of the goal strings from where it was suspended and proceed with the selection algorithm as described earlier. However we now need to examine the rules in the coarse set only. Note that we have already completed the first match phase for all these rules. So this step can be now be skipped by the selection algorithm.

Finally some remarks about efficiency. Suppose there are n rules in a Prolog program. If only m of these rules are included in the coarsely filtered set then computing this set at run time requires at most $O(m)$ time over and above the time required to scan g_1. We have thus managed to exclude the remaining $n - m$ rules without even examining them.

5 Selection using Linear Space

Recall that the space needed for the automaton can become quadratic in the size of the rules. We now briefly outline modifications to our selection algorithm that will reduce quadratic space to linear at the expense of increasing the constant in its running time.

Notice that the same automaton was used to handle the two generic string matching questions, namely, occurrences of prefixes of rule strings in goal strings and vice versa. Recall that to answer the former question we only need the rule strings in the automaton (and not their suffixes) and therefore in such a case the automaton requires only linear space. The idea now is not to use the automaton for dealing with the latter question, viz., prefixes of goal strings in rule strings but to use a *suffix tree* instead.

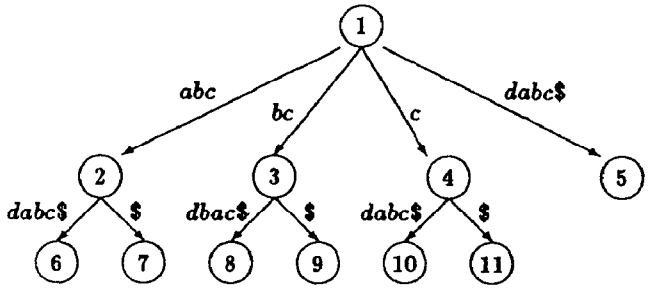


Fig 7: Suffix tree for *abcdabc*

A suffix tree of a string s is a trie in which each root-to-leaf path spells out a distinct suffix of s. Fig.7 is the suffix tree for string $s = abcdabc$. We preprocess all the rule strings and build a suffix tree. Such a tree requires space linear in the size of the rules[7]. In addition to scanning the goal strings with the automaton we now scan them with the suffix tree also. While scanning with the suffix tree its nodes serve as states and the edges as goto transitions. A boolean flag (initialized to false) is maintained with each goal symbol. This flag is set to true if a successful transition is made upon inspecting the goal symbol with the suffix tree. In such a case the node number (from which a transition is made) is also stored with the inspected symbol. To verify whether a prefix α of the goal string matches the substring β of a rule string (see fig.5(b)) we proceed as follows. Suppose $\beta = \alpha$. Then let s_α denote the node reached in the suffix tree upon scanning α. Observe that β is a prefix of a suffix ω. Therefore the root-to-leaf path in the suffix tree that spells out ω must pass through s_α. All we need to do now is verify whether the leaf node of this root-to-leaf path occurs in the subtree rooted at s_α. This again can be done in $O(1)$ time.

Finally, note that each rule string appears twice - once in the automaton and once in suffix tree. However observe that every rule corresponding to a ground fact gives rise to only one rule string. Selecting a fact only involves verifying whether goal strings occur in the fact's rule string. This can be easily answered using the suffix tree alone. So rule strings corresponding to ground facts need not be a part of the automaton. This can result in considerable savings in both space required and running time of applications that deal with voluminous amount of ground facts such as Prolog databases.

6 Indexing Technique using Bit Strings

We now describe another indexing technique suitable for a machine model in which bit string operations of union and intersection are done in constant time. As long as the number of rules with the same predicate name does not exceed the wordsize then this method is of practical importance. (Thus it is not appropriate for important applications such as Prolog databases as they typically involve voluminous number of rules with the same head symbol.)

Each rule tree is again preprocessed into a trie and the tries of all the rules are then merged. The trie for a rule tree is constructed as follows. First we assign a integer label to every edge in the tree. Specifically, if a node v is labelled with a functor f of arity k then v has k subtrees and the edge leading into the i^{th} subtree is labelled i. Next we remove the variable names from all those nodes labelled with variables. Finally every node labelled with a functor symbol is split into two

nodes connected by an edge that is labelled with the original node label. (See trie for rule r_1 in fig.8(a).) The tries for all the rules are then merged together by a process similar to that used in the construction of the automaton in section 3. (Fig.8(b) is the trie for r_1 and r_2). The space required for the trie is linear in the size of the rules.

Let t_i denote the trie for rule r_i and T the trie obtained by merging all the t_i's together. With each node v in T we maintain two sets S_v and M_v. We include r_i in S_v if the path from the root of T to v is a proper prefix of a root-to-leaf path in t_i. If they are identical then r_i is included in M_v instead of S_v (see fig.8(b)).

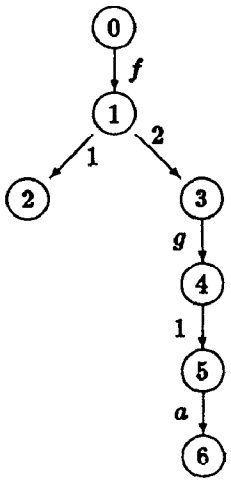
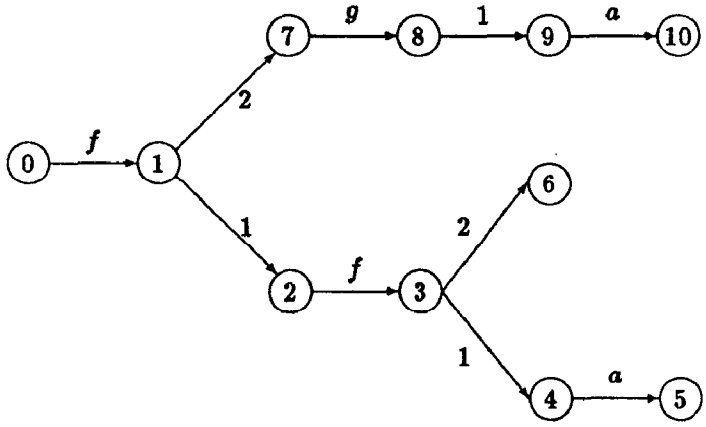


Fig. 8a: trie for r_1



$S_1 = \{r_1, r_2\}$
$S_2 = S_3 = S_4 = \{r_2\}$
$S_5 = S_6 = S_8 = \emptyset$
$S_7 = S_8 = S_9 = \{r_1\}$

$M_1 = M_3 = M_4 = \emptyset$
$M_8 = M_9 = \emptyset$
$M_2 = M_{10} = \{r_1\}$
$M_5 = M_6 = M_7 = \{r_2\}$

Fig. 8b

The goal is scanned in conjunction with T in a recursive fashion. Each recursive call has three parameters - the node u in goal currently being inspected, a node v in T and a set S of rules whose root-to-leaf paths have been successfully matched so far. For the very first recursive call u is the root of the goal tree, v is the root of T and S consists of all the rules whose tries have been merged to form T. The call begins by inspecting the label of u. Suppose u is labelled with a variable then this call returns successfully with S as the set of rules selected at this point. On the other hand if u is labelled with a functor symbol and there is no edge leaving v that is labelled with this functor symbol then it means that prefixes of root-to-leaf paths of rules in S_v that have been matched so far cannot be extended any further and are to be removed from S. Therefore this call returns with $S = S \cap \overline{S_v}$ where $\overline{S_v}$ is set compliment of S_v. Suppose there is an edge from v to w that has the same functor label then we descend to w in T as we have now been able to extend prefixes of root-to-leaf paths of some of the rules in S. Note that these rules must also be present in S_w. However S_w may also have rules that are not in S. Moreover for some rules in S their root-to-leaf paths might have terminated at v. Such rules must be present in M_v. Therefore we create two new sets $S_1 = S \cap S_w$ and $M_1 = S \cap M_v$. Note that for any rule in S_1 we have matched only a prefix of a root-to-leaf path and in order to complete this match we must scan the goal further. To do this we initiate a number of recursive calls as follows. Observe that w has the same number of children as v and the edges leaving w are all labelled with integers. Let $w_1, w_2, \ldots, w_l$ and $u_1, u_2, \ldots u_l$ denote the children of w and u respectively. We then initiate l recursive calls with u_i, w_i, S_1 as the input to the i^{th} call. On returning from this call S_1 is updated to become $S_1 \cap S_i$. Finally, on returning from the l^{th} recursive call S is updated to become $S_1 \cup M_1$. When the first recursive call initiated at the root of the goal is complete then S denotes the set of selected rules. Notice that by associating a bit per rule set intersection and union can be done in $O(1)$

time. In such a case the running time is bounded by the number of recursive calls which in turn is proportional to the size of the goal. However this method does not compute substitutions for variables.

Lastly we briefly mention a modification to the method when set unions and intersections cannot be performed using bit strings. For every rule, S_v has a count of the number of path strings in the rule that passes through v. Similarly M_v has a count of the number of path strings that terminate on v. We associate a counter with every rule. While scanning, this counter gets updated to reflect the number of path strings of the rule that have been matched so far. Upon completion of the scan a rule is selected if its counter value equals the number of path strings in it. An idea similar to this was used in [4] for doing tree pattern matching. Observe that selecting a rule will now require time proportional to the number of leaves in it. In contrast, recall that in our first indexing method a rule is selected in time proportional to the number of substitutions computed and this number is *always less than or equal* to the number of leaves in the rule.

7 Concluding Remarks

In this paper we described an automata-driven indexing technique for Prolog that selects rules quickly at run time. Although we compile all the rule strings in this technique, in practice we can choose the appropriate strings to index on a few arguments only. Thus our technique generalizes known techniques that index on functors. For instance, most existing Prolog implementations index on the outermost functor of specified argument(s), say the i^{th} argument. This involves building hash tables that groups all the rules with the same outermost functor in the i^{th} argument into a single set. To index only on the i^{th} argument using our technique we proceed as follows. As part of compilation we first transform every rule so that its i^{th} argument becomes the first argument. We perform a similar transformation on the goal (prior to execution). Next we construct the coarse filter based on the transformed rules and goal. However we now suspend the scan on examining the second symbol in the goal's first string. This corresponds to the outermost symbol in the goal's i^{th} argument before transformation. If A is the state of the automaton upon reading the second symbol then $S_2 = D_A$ and $S_1 \bigcup S_2$ at this point is exactly the set of rules in the hash table corresponding to the outermost symbol in the goal's i^{th} argument.

We also described another efficient indexing technique that is useful in situations where the number of rules with the same root symbol does not exceed a word-size. This can be of practical importance for small Prolog programs.

References

[1] A.V. Aho and M.J. Corasick, Efficient String Matching: An Aid to Bibliographic Search, CACM, Vol 18 No. 6, June 1975, pp. 333-340.

[2] K.L. Clark and F.G. McCabe, The Control Facilities in IC-PROLOG, Expert Systems in Micro Electronic Age, Ed. D. Michie, Edinburg University Press, 1979.

[3] B. Demoen, A. Marien and A. Callebaut, Indexing Prolog Clauses, To appear in North American Conference in Logic Programming, Cleveland, Oct 1989.

[4] C.M. Hoffmann and M.J. O'Donnell, Pattern Matching in Trees, JACM 29, 1, 1982 pp. 68-95.

[5] D.E. Knuth, J.H. Morris and V.R. Pratt, Fast Pattern Matching in Strings, SIAM Journal of Computing Vol 6, No 2, 1977, pp. 323-350.

[6] A. Martelli and U. Montanari, An Efficient Unification Algorithm, ACM TOPLAS, Vol 4, No. 2, Apr 1982, pp. 258-282.

[7] E. M. McCreight, A Space-Economical Suffix Tree Construction Algorithm, Journal of ACM, Vol 23, No. 2, April 1976, pp. 263-272.

[8] M.S. Paterson and M.N. Wegman, Linear Unification, Journal of Computer System and Science, Vol 16, No. 2, April 1978, pp. 158-167

[9] Quintus Prolog Users Guide, Quintus Computer Systems Inc., Mountain View, California.

[10] K. Ramamohanarao and J. Shepherd, A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases, Proceedings of the Third International Conference on Logic Programming, Jul 1986, Lecture Notes on Computer Science, Vol. 225, Springer Verlag pp. 569-576.

[11] S.K. Debray, The SB-Prolog System, Version 2.3.2: A User Manual, Technical Report 87-15, Department of Computer Science, University of Arizona, Toucson, Dec 1987.

[12] D.H.D. Warren, An Abstract Prolog Instruction Set, Technical Note 309, SRI International.

[13] D.H.D. Warren, Implementing Prolog - Compiling Predicate Logic Programs, D.A.I Research Reports 39, 40, University of Edinburg, 1977.

[14] M.J. Wise and D.M.W. Powers, Indexing PROLOG Clauses via Superimposed Code Words and Field Encoded Words, Proceedings of the IEEE Conference on Logic Programming, Jan 1984, pp. 203-210.