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, doing so can cause an enormous increase in code size and running time if indexing is done naively. This paper describes new and efficient indexing techniques that utilize all the functors in the clause heads and the goal. The salient feature of these techniques is that the selected clause head unifies (modulo nonlinearity) with the goal. In unification (modulo nonlinearity) all the variables in the terms being unified are assumed to be unique, so the only operation performed is of matching their constant portions. So use of our indexing techniques can result in sharper discrimination, fewer choice points, and reduced backtracking. These techniques have been incorporated into a Prolog compiler and using this compiler the run-time performance of a broad spectrum of Prolog programs has been improved.
1. INTRODUCTION

The fundamental computational step in the execution of Prolog programs is the selection and unification of clause heads with the 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.

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 heads 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) [18], Quintus Prolog [14], Stony Brook Prolog [17], and several other Prolog systems [4, 19]. The problems with this approach are that first 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 $p(f(a, c), d)$. Second, 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 (e.g., as done in BIM Prolog [3] and the XSB system [21]). 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 ORing 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 [15]. 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, they are assigned the same code. Note that this method may be suited for applications where arguments are atomic such as databases. 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 that is 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 the running time if indexing is done naively. A “good” indexing technique, therefore, should be able to utilize all the (nonvariable) symbols in clause heads without losing significant structural information and without compromising speed also. The design of such an indexing technique has remained an open problem and we address it in this paper.
1.1. Overview and Summary of Results

In this paper, we describe two new indexing techniques. A salient feature of both techniques is that the selected clause head unifies (modulo nonlinearity) with the goal. We say that a clause head unifies (modulo nonlinearity) with the goal iff it unifies with the goal after uniquely renaming multiple occurrences of variables in both the clause head and the goal. Therefore, a clause head selected by our indexing techniques may fail to unify with the goal iff there are multiple occurrences of a variable either in the clause head or in the goal. Indexing based on unification (modulo nonlinearity) overcomes the shortcomings of indexing based on the first argument mentioned earlier. It also overcomes the main drawback of indexing based on encoding terms since structurally different clause heads containing the same functors (such as \( p(a, b) \) and \( p(b, a) \)) are now distinguished.

Note that the fast unification algorithms of Paterson and Wegman [13] and Martelli and Montanari [9] can be used to perform unification (modulo nonlinearity). The main problem in doing so is that when there are several clause-heads to be selected, the symbols in the goal may need to be reinspected several times. This is quite wasteful and not appropriate for fast indexing.

In our first technique, each clause head is transformed into a set of strings by doing a left-to-right preorder traversal and removing the variables. Thus \( f(a, g(X, b)) \) is transformed into \( fag \) and \( b \). Observe that the clause-head strings so obtained contain all the nonvariable symbols in the head. These clause-head strings are then compiled into a string-matching automaton. At runtime the goal is scanned and the state transitions made by the automaton are recorded. The information embodied in these states is now used to avoid reinspection of symbols in the goal and thereby improve the running time of the technique.

In addition to the above algorithm, which we will refer to as indexing based on string-matching, we describe another indexing method based on tries using a bit-vector model. Herein sets of clause heads are represented by bit vectors, and intersection and union operations on them are assumed to require constant time. This method, which we will refer to as trie based indexing, is suited for Prolog programs in which the number of clause heads with the same predicate name does not exceed the size of a constant number of words.

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. Often clause selection in typical Prolog programs involves linear clause heads and goal.\(^1\) Since we perform unification modulo nonlinearity, this fact can be gainfully exploited by our technique to obtain the unifier during the indexing process. In contrast, indexing methods employed in almost all extant Prolog compilers, such as Quintus Prolog, ALS Prolog, and SB Prolog, 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 [13, 10]).

The following is a brief summary of the results in this paper.

1.1.1. INDEXING BASED ON STRING-MATCHING

\(^1\)In a linear clause head, each variable occurs only once.
The string-matching automaton is constructed at compile time by preprocessing all the clause heads. At run time the goal is scanned once prior to selection of any clause head. Suppose there are \( m \) clause heads with the same predicate symbol. Let \( m_i, l_i, \) and \( k_i \) denote the size, total number of leaf nodes, and the total number of variable occurrences in the \( i \)th clause head, respectively. Similarly let \( n, l_g, \) and \( k_g \) denote the size, total number of leaf nodes, and total number of variable occurrences in the goal, respectively. We show that the worst-case running time of our indexing technique is \( O(\sum_{i=1}^m \min\{l_i, k_i + k_g\} + n) \). In contrast, using the fast unification algorithms in [13, 10] would have resulted in \( O(\sum_{i=1}^m \min\{m_i, n\}) \) time complexity for indexing. (Note \( \min\{l_i, k_i + k_g\} \leq \min\{m_i, n\} \).)

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 the indexing technique.

Scanning the goal a priori before selection may result in examination of symbols in the goal that are never used in any string-matching operation. We significantly modify our algorithm so that the goal symbols are scanned on demand, i.e., only when needed for a string-matching operation. Demand-driven indexing also improves the run-time performance. In particular, we show that it is possible to reduce both \( n \) and \( l_g \) in the complexity figure above.

1.1.2. TRIE BASED INDEXING

A trie is constructed at compile time by preprocessing all the clause heads. The time to construct this trie and its space requirements are both linear in the size of the clause heads.

Sets of clause heads with the same predicate symbol are represented by bit vectors. The goal is scanned in conjunction with the trie. Each symbol of the goal is seen only once. Furthermore, each node in the trie is visited at most once. At each of these nodes visited an AND and/or an OR operation is performed on these sets. At the end of the scan, the bits that are set denote the clause heads that unify (modulo nonlinearity) with the goal. The worst-case running time of this algorithm is \( O(n) \).

1.1.3. IMPLEMENTATION

We recently completed the design and implementation of the \( \nu \)-ALS compiler that incorporates these indexing algorithms [5]. Using it we have enhanced the performance of a broad spectrum of Prolog programs ranging from small procedures with a few shallow rules to complex procedures with deep structures. Our experience with \( \nu \)-ALS strongly suggests that indexing based on unification (modulo nonlinearity) is a viable idea in practice and that a broad spectrum of realistic programs can realize all of its benefits.

This paper is organized as follows. Section 2 contains the description of our string-matching based indexing algorithm. We begin this section with a simple indexing algorithm that forms the framework for our efficient algorithms. Using

\[ ^2 \text{A clause head can be viewed as a labeled tree.} \]
this simple algorithm, we outline the issues crucial to compiling clause heads for fast indexing (Sections 2.2 and 2.3). In Section 2.4, we describe the compilation of clause heads into a finite-state string-matching automaton. Section 2.5 contains detailed description of indexing based on string-matching. A technique to further fine tune our algorithm appears in Section 2.6. Section 3 contains the details of demand-driven algorithm. In addition, it also outlines a technique to reduce the space requirements of the string-matching automaton. Section 4 describes indexing based on the bit-vector model. In Section 5, we summarize the implementation details of the $\nu$-ALS compiler and the performance of Prolog programs compiled using it. Finally, the concluding remarks appear in Section 6 and this is followed by appendixes that contain the proof of correctness and complexity analysis of our algorithms.

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.

2. INDEXING BASED ON STRING-MATCHING

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

2.1. Simple Indexing Algorithm
Our indexing 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

3 We assume that functors have unique arity. In cases when this is not satisfied, functors together with their arities can be compared in equality tests and thus the scope of the results in this paper are not affected by this assumption.


```plaintext
fail := false;
repeat
  if $G(p_g)$ and $R(p_r)$ are both functors then
    begin
      if $G(p_g) \neq R(p_r)$ 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 = true) or (G and R are completely scanned)
```

FIGURE 2. Simple selection algorithm.

if and only if it unifies with the goal after uniquely renaming multiple occurrences of variables in both the rule and the goal. The structure of rules selected by our indexing 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.

Figure 2 is an outline of such a indexing 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 [13, 10]).

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 indexing 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 1. 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 labeled with a variable, say $X$, and $p_r$ to some node, say $v$. Then $p_g$ is advanced by 1, 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. Toward 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 preorder as goal strings and rule’s preorder as rule strings (see Figure 1).

Observe that the two variables, for which substitutions are computed in the skip phases immediately preceding and succeeding a match phase, are either both rule variables (such as \( X_i, X_{i+1} \) in Figure 3a) or are both goal variables (such as \( Y_j, Y_{j+1} \) in Figure 3c) or one is a goal variable and the other is a rule variable (such as \( X_i, Y_{j+1} \) in Fig. 3b and \( Y_j, X_{i+1} \) in Figure 3d). Clearly these are the only possible situations. In Figure 3, \( \alpha \) and \( \beta \) 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 \( p \) and \( q \) for \( X_i \) in Figure 3a).

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

1. Does \( \alpha \) (rule string) occur in \( \beta \) (goal string) at \( p \)? (Figure 3(a).)
2. Does a prefix of \( \alpha \) match a suffix of \( \beta \) at \( p \)? (Figure 3(b).)
3. Does \( \beta \) (goal string) occur in \( \alpha \) (rule string) at \( p \)? (Figure 3(c).)
4. Does a prefix of $\beta$ match a suffix of $\alpha$ at position $p$? (Figure 3d.)

Observe that these four questions are a special case of the following two generic questions. Given a specific position $p$:

- **Q1.** Does the prefix of a rule string occur in a goal string at position $p$?
- **Q2.** Does the prefix of a goal string occur in a rule string at position $p$?

We now show how to compile the rule strings into a finite-state string-matching automaton that at run time will enable us to answer these questions in $O(1)$ time.

### 2.4. Compilation of Rules

Central to our technique is a finite-state automaton that is constructed from the rule strings. We use the Aho and Corasick (see [1] for details) algorithm to construct such an automaton. Following [1] we refer to the strings recognized by the automaton as the **keywords** of the automaton.

The automaton consists of nodes called **states** and two types of links—**goto** and **failure**. The goto links are labeled with symbols from the alphabet of the keywords. These links together with the states form a trie structure known as the **goto tree**, whose root is the start state (see Figure 4 for illustration). Following [1], we say string $\lambda$ represents state $\gamma$ if the path in the trie from the start state (the root node) to state $\gamma$ spells out $\lambda$. The construction using the Aho and Corasick algorithm ensures that every keyword is represented by a state in the automaton. This implies that every prefix of a keyword is also represented by some state in the automaton. In fact, there is a one-to-one correspondence between the states of the automaton and unique prefixes of keywords.

The automaton scans the input text for recognizing occurrences of keywords. While scanning, it makes either a goto or a failure transition. Suppose the automaton is in state $u$ after scanning the first $j$ symbols of the input $a_1 a_2 \cdots a_j a_{j+1} \cdots a_n$. If there is a goto link labeled $a_{j+1}$ from $u$ to $w$, then the automaton makes a goto transition to $w$. Now, we can state the following lemma.
Lemma 2.1 (Aho and Corasick). The string represented by $w$ is the longest suffix of $a_1a_2 \cdots a_{j+1}$ that is also a prefix of some keyword.

On the other hand, if there is no such link labeled $a_{j+1}$ from $u$, then it makes a failure transition. If this transition takes the automaton to a state $v$, then the following lemma applies.

Lemma 2.2 (Aho and Corasick). The string represented by $v$ is the longest proper suffix (among the strings represented) of the string represented by $u$.

We refer to $v$ as the failstate of $u$. Suppose $a_{j+1}$ is such that the automaton is still unable to make goto transitions from $v$ with $a_{j+1}$. Then it again makes a failure transition and continues to do so until it reaches a state from which it can make a goto transition with $a_{j+1}$. Since the start state has goto links for all symbols in the alphabet, the automaton is able to make (eventually) a goto transition on every symbol of the input.

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 string occurs in a goal string and vice versa.

We now extend the automaton to handle such questions. For clarity of notation we will implicitly assume the presence of position $p$ in every instance of $Q1$ and $Q2$. The rule strings of the clause heads in the Prolog program form the keywords of this automaton. At run time the automaton scans the symbols of the goal strings. Suppose we want to know whether the prefix $\alpha$ of a rule string, say $r_i$, matches the substring $\beta$ of goal string (see Figure 5a).

Let $s_\beta$ denote the state of the automaton after reading the last symbol in $\beta$. If $s_\alpha$ is the state representing $\alpha$, then the following theorem is applicable.

Theorem 2.1. $\alpha$ matches $\beta$ iff $s_\alpha$ is reachable from $s_\beta$ through zero or more failure transitions only.

PROOF. Straightforward from Lemmas 2.1 and 2.2. \(\Box\)

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. (Figure 4(b) is the fail tree for the automaton in Figure

---

A ground term has no variables.
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 \(\alpha\) to match \(\beta\), \(s_\alpha\) must be an ancestor of \(s_\beta\) 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 (which is \(Q_1\)).

Now we extend the automaton to answer \(Q_2\), i.e., whether a prefix \(\alpha\) of a goal string matches a substring \(\beta\) of a rule string (see Figure 5(b)). \(s_\alpha\) is the state of the automaton on scanning \(\alpha\) and \(s_\beta\) is the state corresponding to prefix \(\gamma\) of the rule string. Note that the goal strings are not available at compile time. Because 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 2.1 is no longer useful to answer questions related to prefixes of goal strings. Specifically, even if \(\alpha\) does not match \(\beta\), \(s_\alpha\) can still be an ancestor of \(s_\beta\) in the fail tree. For instance, the automaton in Figure 4(a) ends up in state 1 upon scanning the prefix \(gf\) of a goal string. Now observe that even though state 1 is an ancestor of state 2 in the fail tree, \(gf\) does not occur in the rule string \(ffa\). The main problem is that the string represented by \(s_\alpha\) (which is \(f\) above) is only a proper suffix of \(\alpha\) (which is \(gf\) above) and \(s_\alpha\) is an ancestor of \(s_\beta\) whenever the string represented by \(s_\alpha\) is a suffix of the string represented by \(s_\beta\). In the above example, \(s_\alpha\) is state 1 and \(s_\beta\) is state 2, which represents \(ff\). Had we also used the string \(gf\) as one of the keywords of the automaton, then on scanning \(gf\), the automaton would have reached a state \(s_6\) that must represent \(gf\). In such a case, \(s_6\) could never have become an ancestor of state 2. The solution now is to 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 Figure 5(b), suppose \(\alpha\) matches \(\beta\). Now observe that \(\beta\) is a prefix of \(\omega\) 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 Figure 1, we now insert their suffixes $a, fa$ also into the automaton, in Figure 4a, as its keywords. Figure 6a is the resulting automaton.) Now Theorem 2.1 can be used to correctly answer instances of Q2 whenever prefixes of goal strings do occur in some rule strings. However, we still need to handle instances of Q2 when a goal prefix does not occur in any rule string. Suppose $\delta$ is a goal prefix that does not occur in any rule string. On scanning it, the automaton ends up in a state $s_\delta$ that does not represent $\delta$, and so if Theorem 2.1 is used in such cases, then we can erroneously conclude that $\delta$ indeed occurs in a rule's substring. Fortunately, there can be no state in the automaton that can represent $\delta$ and hence $s_\delta$ can only represent a proper suffix of $\delta$. This implies that the depth of $s_\delta$ in the goto trie must differ from that of $|\delta|$. Based on this observation, we can answer all instances of Q2 regardless of the goal prefix involved. This is reflected by the following theorem.

**Theorem 2.2.** If $\alpha$ is a prefix of a goal string and $\beta$ is a rule's substring as shown in Figure 5(b), then $\alpha$ matches $\beta$ iff $s_\alpha$ is an ancestor of $s_\beta$ and depth $(s_\alpha)$ in the goto tree $= |\alpha|$.

**Proof.** $\implies$ That is, $\alpha$ matches $\beta$. Since $\beta$ is a prefix of $\omega$, so is $\alpha$. Since $\omega$ is a keyword of the automaton, $s_\alpha$ will represent $\alpha$. Therefore, depth $(s_\alpha) = |\alpha|$. Observe that $\beta$ is a suffix of $\gamma$ and so is $\alpha$. Also observe that $s_\beta$ represents $\gamma$ and so $s_\alpha$ is an ancestor of $s_\beta$ by Lemma 2.2.

$\impliedby$ Since $s_\alpha$ is an ancestor of $s_\beta$, the string represented by $s_\alpha$ is a suffix of $\gamma$, but depth $(s_\alpha) = |\alpha|$. Therefore, the string represented by $s_\alpha$ must be $\alpha$ by Lemma 2.1. Hence, $\alpha$ is a suffix of $\gamma$ and $\beta$ in fact is this suffix. $\square$

Observe that the depths of all states in the goto trie can be computed at compile time by preorder traversal. With the depth information readily available, the condition of theorem 2.2 is verifiable in $O(1)$ time. Thus all string-matching questions can be answered in $O(1)$ time.

Finally note that the size of 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).

2.5. Algorithmic Details

Based on the discussions of the previous sections, we now present the algorithmic details of an efficient algorithm (given below as procedure Index) for indexing based on the string-matching automaton.

The rules are preprocessed at compile time to construct the Aho–Corasick automaton and its fail tree. Procedure Index uses two arrays of records $R$ and $G$ that contain information about the symbols scanned by a preorder traversal of rule $r$ and the goal $g$, respectively. Each record in $R$ has four fields: label, varposn, subtree, and state. The label field is used to specify the functor/variable symbols. The varposn field at $R[i]$ is set to the preorder number of the nearest variable node that appears after $i$ in preorder. The subtree field of $R[i]$ is set to the preorder number of the last node in the subtree rooted at node $i$. The state field specifies the state of the automata reached on reading the symbol at $i$ while scanning $R$. The structure of array $G$ is identical to $R$. In addition, the algorithm uses $p_g$, $p_r$, $l_g$, $l_r$, $p_g$, $p_r$, $l_g$, $l_r$, 

...
and $v_g$. $p_g$ and $p_r$ point to positions in $G$ and $R$, respectively, up to which the algorithm has proceeded without failure. $l_g$ and $l_r$ are the lengths of remaining portions of goal and rule strings from $p_g$ and $p_r$, respectively. $v_g$ is set to true if the immediately preceding substitution was made to a goal variable. $pf$ and $nd$ are functions that return the preorder number and the number of descendants of a state in the fail tree, respectively, whereas function $depth$ returns the depth of a state in the goto tree.

At run time the goal tree is scanned prior to selection of any rule and all the fields in each record of $G$ are filled. (Note that $R$ is filled at compile time.) Now procedure Index is then invoked to select $r$.

**Procedure Index**
BEGIN
1. $fail := FALSE$;
2. /* Perform first string match operation */
3. $p_g := p_r := 1$;
4. $l_g := G[p_g].varposn; l_r := R[p_r].varposn$ /* $l_g$, $l_r$ are lengths of goal and rule strings resp.*/
5. IF $l_g > l_r$ THEN
6. /* goal string is longer */
7. $fail := G[l_r].state \neq R[l_r].state$;
8. $p_r := p_g := l_r + 1$;
9. ELSE
10. $fail := G[l_g].state \neq R[l_g].state$;
11. $p_r := p_g := l_g + 1$;
12. WHILE $fail = FALSE \land (G$ and $R$ are not completely scanned) DO
13. WHILE $G[p_g].label$ or $R[p_r].label$ is a variable DO
14. IF $G[p_g].label$ is a variable THEN
15. $v_g := TRUE$;
16. $p_g := p_g + 1$;
17. $p_r := R[p_r].subtree + 1$;
18. ELSE
19. $v_g := FALSE$;
20. $p_r := p_r + 1$;
21. $p_g := G[p_g].subtree + 1$;
22. ENDIF;
23. END;
24. IF $G$ and $R$ are not completely scanned THEN
25. /* Both $p_g$ and $p_r$ point to functor nodes and $v_g$ specifies whether the immediately preceding substitution is for rule or goal variable */
26. $l_g := G[p_g].varposn - p_g + 1$;
27. $l_r := R[p_r].varposn - p_r + 1$;
28. IF $-v_g$ THEN /* Check occurrence of prefix of rule string in current goal string */
29. IF $l_g > l_r$ THEN /* Check occurrence of entire rule string in goal string as in Figure 3a */
30. $pre_g := pf (G[p_g + l - 1].state);$  
31. $pre_r := pf (R[p_r + l - 1].state);$
The proof of correctness of procedure Index and complexity of its running time appears in Appendix A. Herein we only state the main results (see Theorems A.1 and A.2 for details).

**Theorem 2.3 (Correctness).** \( r \) unifies with goal (modulo nonlinearity) iff procedure Index terminates with \( \text{fail} = \text{false} \).

Let \( l_g \) and \( k_g \) be the number of leaves and number of variable occurrences in the goal. Similarly, let \( l \) and \( k \) denote the number of leaves and number of variable occurrences in rule \( r \). Then the following theorem can be stated.

**Theorem 2.4.** The worst-case time required by procedure Index to select rule head \( r \) is \( O(\min\{k + k_g, l, l_g\}) \).
2.6. Improving Selection

A straightforward way to perform rule selection is to invoke procedure Index once for every rule. However, such a method regards every rule as a likely candidate for inclusion in the filtered set. Thus it unnecessarily examines 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 $S_1 = \{ r | \text{the first string of } r \text{ is a prefix of } g_1 \}$ and $S_2 = \{ r | g_1 \text{ is a prefix of the first string of } r \}$. The following is a description of the ideas underlying the construction of the coarsely filtered set $S_1 \cup S_2$. First we need the following concept. We say that $A$ is the primary accepting state of a keyword string $s$ if it is both an accepting state for $s$ and represents $s$. For instance, in Figure 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 $\beta$ matches a prefix $\alpha$ of $g_1$. Since $\beta$ 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 $\beta$. This implies that $\alpha$ 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 $\beta$, 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 $\beta$. 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 $\beta$, then the primary accepting state of $\beta$ 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 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.

3. REFINEMENTS TO INDEXING BASED ON STRING-MATCHING

We now describe two optimizations for the automata-driven indexing algorithm.
The first one reduces the number of goal symbols examined during selection. The
second optimization preprocesses the rules using linear space at the expense of
increasing the running time by a constant factor.

3.1. Demand-Driven Indexing

Observe that the selection algorithm described in Section 2.5 scans the goal prior to
calling procedure Index. Scanning the goal a priori can result in inspecting symbols
that are not needed for selecting any of the rules. Specifically, inspecting subtrees
of goal that occur within a variable substitution of every rule is unnecessary. For
example, in Figure 1 goal's subtree \(g(a)\) occurs within the substitution for \(X\) (in
rule 1) as well as \(Y_1\) (in rule 2). Note that the simple selection algorithm (see Figure
2) will inspect a goal symbol iff it is needed to accept or reject one or more rules.

We now outline the main ideas involved to avoid inspection of any goal symbol
that is not needed for selecting or rejecting any rule. The approach is to scan the
goal on demand as follows. All the rules not yet rejected are in two states—active
and suspended. The next symbol of the goal is scanned only when it is needed to
determine the outcome of the string match to be performed on behalf of an active
rule whose selection process is currently in progress. Specifically, suppose \(r\) is such
a rule. Further suppose the next string match operation for \(r\) requires comparing
the next \(m\) symbols in the goal. Assume that only \(n < m\) of these symbols have
been scanned and all these \(n\) symbols match the corresponding symbols in the
rule string. In such a case the \((n + 1)\)th symbol must be scanned. If this symbol
matches the corresponding symbol in \(r\), then the scan will continue until either a
mismatch is detected or all the \(m\) symbols match. If there is a mismatch, then
rule \(r\) is rejected and another active rule \(q\) is picked. The states of the automaton
are recorded whenever a goal symbol is scanned. These states can now be used
to perform string-matching operations of \(q\) (or any other rule), involving only the
scanned symbols, in \(O(1)\) time.

Now consider the case when all the \(m\) scanned symbols in the goal have matched
with those in \(r\). In such a case, either the \((m + 1)\)th symbol in the goal or the
corresponding symbol in the rule must be a variable. In the former case, a substi-
tution is computed for the goal variable and scanning continues on behalf of the
next string match needed for selection of \(r\). This will once again involve demand-
driven scanning of the next goal string that begins at the \((m + 2)\)th symbol. (Note
that the automaton will always scan any new goal string of the goal from the start
state as in this case.) On the other hand, suppose the symbol in r, corresponding
to the \((m + 1)\)th symbol in goal, is a variable and is the label of node \(v\). Let the
\((m + 1)\)th symbol in the goal be the label of \(u\). The subtree rooted at \(u\) is the
substitution for this variable in \(r\). This subtree has not yet been scanned. The
next string-matching operation for \(r\) will only require scanning nodes appearing
after those in the subtree rooted at \(u\) in preorder. Although the symbols in this
subtree are not needed for the selection of \(r\), they may be needed for selection of
some other rule, say \(q\). Note that the scanned symbols that are needed to perform
a string-matching operation for at least one rule must be stored contiguously in an
array. Furthermore, the sequence of nodes appearing in this portion of the array
must have successive preorder numbers. This means that if \(q\) needs the \((m + 1)\)th
symbol in a string-matching operation, then it must appear immediately after the
\(m\)th symbol in the array. Hence we cannot scan the goal after skipping the subtree
at \(u\). Therefore, we suspend \(r\) on the goal node that immediately follows the last
node in the subtree rooted at \(u\) in preorder. We then pick another active rule and
proceed with the selection of this rule as explained above. If in this process the
node on which \(r\) is suspended is scanned, then \(r\) is activated. On the other hand,
suppose this node is not yet examined and there is no active rule. Then it means all
the rules are suspended. In such a case, we reactivate the rule that is suspended on
the node that is farthest from the root of the goal tree. This is because the subtree
skipped by such a rule is not needed by any rule for string-matching purposes.

We illustrate these ideas on the goal and the rules in Figure 1 using the automa-
ton in Figure 6. 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. Because 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 because the state representing its first
string and the state of the automaton on reading the first goal symbol are the same
(Figure 7(a) shows the contents of the goal array on inspecting its first symbol).
The next phase is a skip (triggered\(^5\) by \(X\)). The subtree rooted at 2 in the goal is
skipped 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,

\(^5\)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.
we need to scan the goal further because the symbols inspected so far (which is 1) is shorter than that required to perform this step. We scan two more symbols (see Figure 7(b)).

The first string match of $r_2$ succeeds because the state representing its first string $ffa$ is the same as that in the third 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 Figure 7(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 the 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 because the node on which it was suspended (node 6) has become the root of a substitution. Following this skip phase, $r_2$ is selected because 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 Figure 7(d)). Since 7 is labeled 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 the 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 because 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. Because this is the last node in $r_1$’s preorder, it also is selected.

3.1.1. DEMAND-DRIVEN ALGORITHM. We now present the algorithmic overview of demand-driven indexing. Conceptually, we can view demand-driven indexing as execution of several instances of procedure $Index$ concurrently. Each instance is involved in the selection of one rule. Let $Index_i$ denote the instance involved in the selection of $r_i$. (Table 1 is a summary of global variables used in this algorithm.)

The rules that have not yet been eliminated are either in suspended queue or active queue. Recall that a rule is suspended whenever it skips a subtree of the goal that is not yet scanned completely. $Index_i$ is stopped whenever $r_i$ is suspended and restarted when the selection of $r_i$ is resumed. When $Index_i$ is restarted, it continues from the point where it was stopped. When there are sufficient symbols in the scanned portion of the goal to perform a string-matching operation in $O(1)$ time, $Index_i$ does so. Otherwise, the string-matching operation is performed in two steps. Specifically, suppose the string-matching operation requires $m$ symbols and only $n < m$ symbols are available. Now $Index_i$ first performs a string-matching operation containing the $m$ available symbols in $O(1)$ time. If the outcome is successful, then the remaining symbols are compared one at a time by demand-driven scanning of the goal.

Note that prior to performing a string-matching operation in procedure $Index$,
TABLE 1. Global Variables Used in Demand-Driven Algorithm

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G$</td>
<td>Goal array that is constructed incrementally</td>
</tr>
<tr>
<td>$end_g$</td>
<td>Integer specifying current length of $G$</td>
</tr>
<tr>
<td>$R[1..m]$</td>
<td>$R[i]$ is to rule $r_i$ what $R$ is to rule $r$ in procedure $Index$</td>
</tr>
<tr>
<td>$p_r[1..m]$</td>
<td>$p_r[i]$ is used as an index into $R[i]$ by procedure $Index$, just as $p_r$ indexes $R$ in procedure $Index$</td>
</tr>
<tr>
<td>$p_g[1..m]$</td>
<td>Similar to array $p_r$, but used to index goal array $G$</td>
</tr>
<tr>
<td>$l_g[1..m]$</td>
<td>$l_g[i]$ is the length of remaining portion (from $p_g[i]$) of goal string that can be used in a string-matching operation</td>
</tr>
<tr>
<td>$l_r[1..m]$</td>
<td>Similar to $l_g$ but gives the length of rule strings</td>
</tr>
<tr>
<td>$fail[1..m]$</td>
<td>$fail[i] = false$ implies that $r_i$ is not yet eliminated</td>
</tr>
<tr>
<td>$suspend[1..m]$</td>
<td>$suspend[i] = false$ implies that $r_i$ is active</td>
</tr>
<tr>
<td>$fringestack$</td>
<td>A stack of fringe nodes</td>
</tr>
<tr>
<td>$abort$</td>
<td>A boolean flag when set to true implies no rule is active</td>
</tr>
</tbody>
</table>

we compute the length of the remaining portions of goal and rule strings (lines 4 and 26–27). (This is needed to identify which of the four string-matching operations is required for comparing these strings.) A similar computation needs to be performed in each $Index_i$ prior to a string match. Note that this computation makes use of the $varposn$ field. If the goal is completely scanned prior to start of selection, then the length of each goal string is available and hence the $varposn$ field can be initialized. However, in demand-driven scanning the lengths of goal strings are not available prior to the start of the selection; they are obtained during scanning only. Therefore, the $varposn$ fields in $G$ can only be computed dynamically. Similarly, the $subtree$ field in $G$ used by procedure $Index$ to compute substitutions for rule variables needs to be computed dynamically during the scanning process. In the following text, we describe how these fields will be computed.

We regard each nonoverlapping substrings of goal’s preorder that is used in a string match operation as a goal string. Based on this view, a goal string can terminate in two ways. In the first case, it terminates whenever the next symbol inspected in the goal is a variable. In the second case, the current string terminates when all rules get suspended, implying a portion of the goal tree must be skipped. Whenever a goal string terminates, all records of $G$ belonging to this string will have their $varposn$ field initialized to $end_g + 1$ (see Table 1).

Suppose that a node $v$ in the goal is currently being examined. Assume that the information about $v$ is stored in $G[i]$. Now $G[i].subtree$ points to the last node in the subtree rooted at $v$ that will be examined. This information is easily obtained as follows. A recursive algorithm is used to visit nodes in preorder. When the recursive call started at $v$ terminates, the information about last node visited in the subtree rooted at $v$ is in $G[end_g]$. Hence $G[i].subtree$ will be assigned the value of $end_g$ upon exit from the recursive call started at $v$.

**Definition 3.1 (Fringe node).** We say that a node $v$ in the goal is a fringe node iff $v$ has been visited by the demand-driven scanning and $v$ has at least one child that has not yet been visited.
FIGURE 8. Suffix tree for abcdabc$.

Note that the set of fringe nodes is dynamic. In particular, when the last child of a fringe node is visited, it ceases to be a fringe node from that instance. The set of fringe nodes is kept in fringestack. In addition, we also add the field fringenode to each record in G. Suppose G[x] contains information about node n. Then G[x].fringenode is set to the node on top of the fringe stack when n was visited. This information is needed to suspend a rule that skips the subtree rooted at n. Specifically, if a rule skips the subtree rooted at n, then it will be suspended at the first uninspected child of G[x].fringenode. This is because this child is the preorder successor of the last node in the subtree rooted at n. When G[x].fringenode is visited, this rule moved to the active queue.

The algorithmic details of the demand-driven algorithm and the proof of its correctness are given in Appendix B.

3.2. Selection Using Linear Space

In our preprocessing, we build an Aho-Corasick automaton to recognize occurrences of all the rule strings as well as their suffixes. Therefore, the time and space required for this preprocessing is quadratic in the sum of the sizes of the rule heads. We now briefly outline modifications to our algorithm that will reduce the space and time required to preprocess the rule heads at the expense of increasing the running time by a constant factor.

Notice that we use the same automaton 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 construction of the automaton requires only linear space and time. 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.

A suffix tree of a string s is a trie in which each root-to-leaf path spells out a distinct suffix of s. Figure 8 is the suffix tree for string s = abcdabc$. We construct the suffix tree on the rule strings as follows. First we concatenate the rule strings from all the rule into a single string. Let this string be \( \mathcal{R} \). We preprocess \( \mathcal{R} \) and construct a suffix tree for it. This tree requires \( O(|\mathcal{R}|) \) space and can be constructed in \( O(|\mathcal{R}|) \) time [11]. Now in addition to scanning the goal strings with the automaton, we scan them with the suffix tree also. This scanning differs from that of scanning with the automaton due to the following reasons. First, unlike the
automaton, the edges in the suffix tree are labeled with substrings of $\mathcal{R}$. Second, there are no failure links in a suffix tree.

When scanning with a suffix tree, each node $n$ in the tree is viewed as representing a sequence of states. Specifically, if $n$ is a node in the suffix tree with an edge between it and its parent $m$ labeled with the string $\alpha = a_1a_2\ldots a_i$, then during scanning, $n$ represents a sequence of $l$ states $m = n_1, n_2, \ldots, n_l = n$ with the goto transition between $n_i$ and $n_{i+1}$ labeled with $a_i$. When the goal is now scanned, these states of the suffix tree are stored with the preorder of the goal. For convenience we store the state $n_i$ as a pair $(n, i)$. Suppose we are unable to make a goto transition with the symbol currently seen in the goal string. Then we terminate the scan of this goal string and proceed to scan the next goal string. Although some functors in a goal string now may not have any state of the suffix tree associated with them, we can still answer all the string matching questions involving prefixes of goal strings and substrings of rule strings. The basis for this is the following theorem.

**Theorem 3.2.** If a prefix $\alpha$ of a goal string matches a substring of $\mathcal{R}$, then there are two nodes $n$ and $m$ in the suffix tree $T$ of $\mathcal{R}$ satisfying the following conditions.

1. $n$ is a child of $m$.
2. $\alpha$ is a prefix of the path from root of $T$ to $n$.
3. The path from root of $T$ to $m$ is a prefix of $\alpha$.

**Proof.** In $T$ each root-to-leaf path spells out a distinct suffix of $\mathcal{R}$. Clearly, if $\alpha$ matches a substring of $\mathcal{R}$, it is a prefix of some suffix $\beta$ of $\mathcal{R}$. Let us consider the root-to-leaf path that spells out $\beta$. Because $\alpha$ is a prefix of this path, there should be a node $v$ such that the path of $v$ is the longest prefix of $\alpha$. If $w$ is its child along the path that spells out $\beta$, then $v$ and $w$ satisfy all the conditions of the theorem. □

Theorem 3.1 states that if a prefix $\alpha$ of a goal string matches a substring of a rule string (and hence substring of $\mathcal{R}$), then we will be able to scan this goal string with the suffix tree up to the last functor in $\alpha$. Moreover, if we are unable to scan the $i$th functor in a goal string, then any prefix of this goal string that contains this functor will not match a substring of any rule string. Therefore, any rule selection that involves matching such a prefix to a substring of its preorder should be discarded because unification of this rule with the goal will fail.

Upon completing the scan of a goal string $\omega$, our second generic string-matching question can be answered as follows. Let us assume that in the selection of rule $r_i$, in a match step, we need to compare a prefix $\alpha$ of a goal string $\omega$ with the substring between positions $j_1$ and $j_2$ in rule string $r_i$. Clearly, this substring is a prefix of some suffix $\mathcal{R}_1$ of $\mathcal{R}$. Let the path from root of $T$ to the leaf $v$ spell out $\mathcal{R}_1$. In order to complete this match successfully, $\alpha$ should be a prefix of $\mathcal{R}_1$. In particular, if $(n, i)$ is the state stored with the last functor in $\alpha$, then $n$ should be an ancestor of $v$. This can be easily verified in $O(1)$ time.

By a preorder scan of the suffix tree, we can identify for each suffix of $\mathcal{R}$ the leaf node that represents this suffix. This can be done in time proportional to the number of nodes in the suffix tree which is at most $O(|\mathcal{R}|)$. During the scan we can also compute the preorder number and number of descendants for each node in the suffix tree. This means we can perform all the steps need to answer our second
generic question in $O(1)$ time.

Because we do not include suffixes of all rule strings, the Aho–Corasick automaton can therefore be constructed in linear space and time. The construction of the suffix tree as well as processing it as described above takes only linear space and time. In conclusion, the preprocessing described in this section can be accomplished only using linear space in time proportional to the sum of the sizes of the rules. Upon completion of preprocessing, we can answer all the string-matching questions raised in the selection of rules in constant time.

Note that each rule string appears twice—once in the automaton and once in the 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 its corresponding 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 the space required and the running time in applications that deal with voluminous amount of ground facts such as Prolog databases.

Finally, observe that the ground facts are not part of the automaton and hence our coarse filtering technique cannot be applied to select them. However, it is quite easy to obtain a coarse filter for ground facts using the suffix tree alone. Let $r_1, r_2, \ldots, r_m$ be the set of ground facts. Further, let us call a suffix $R$ of $R$ a key suffix for $r_i$ if the (only) rule string in $r_i$ is $R$'s prefix. By a single scan of the suffix tree, we can compute for each node $v$, a set $C_v$ of ground facts, such that the key suffixes of facts in $C_v$ are represented by (leaves that are) descendants of $v$. Using $C_v$, coarse filter can be constructed as follows. First we scan the goal's first string with the suffix tree. If the scan is complete and we reach node $v$, then the coarse filter is $C_v$; otherwise it is empty.

4. TRIE BASED INDEXING

We now describe another indexing technique suitable for a machine model in which sets of rules can be represented by bit vectors, and set operations such as union and intersection can be carried out in constant time by ANDing and ORing these bit vectors. As long as the number of rules with the same predicate name does not exceed a constant (i.e., the rule set fits in $O(1)$ machine words), this method is of practical importance.

In this method each rule tree is preprocessed into a trie and the tries of all the rules are then merged into a single trie $T$. The trie for a rule tree is constructed as follows. First we assign an integer label to every edge in the tree. Specifically, if a node $v$ is labeled with a functor $f$ of arity $k$, then $v$ has $k$ subtrees and the edge leading into the $i$th subtree is labeled $i$. Next we remove the labels of all those nodes labeled with variables. Finally, every node labeled with a functor symbol is split into two nodes connected by an edge that is labeled with this functor. The tries thus obtained for rules $r_1$ and $r_2$ in Figure 1 are given in Figure 9.

The tries constructed for the rules are merged to obtain $T$ by an iterative algorithm. Let $t_i$ denote the trie of rule $r_i$ and $T_{i-1}$ denote the trie obtained by merging tries $t_1, t_2, \ldots, t_{i-1}$. In the $i$th step of the iteration, $t_i$ and $T_{i-1}$ are merged to obtain $T_i$. Merging $t_i$ and $T_{i-1}$ involves the following two steps. We first obtain the set of root-to-leaf paths in $t_i$. Following this, we insert each path string thus
obtained into $T_{i-1}$ using a technique similar to that used in the construction of the Aho–Corasick automaton [1]. Upon inserting the last path string, we get $T_i$.

With each node $v$ in the trie $T$, 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 $S_v$. The trie obtained by merging $t_1$ and $t_2$ in 9 is given in Figure 10.

Algorithm TrieSel given in Figure 11 performs indexing using $T$. This algorithm is recursive. 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 that have not been eliminated 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

- $S_1 = \{r_1, r_2\}$
- $S_2 = S_3 = S_4 = \{r_2\}$
- $S_5 = S_6 = S_7 = \emptyset$
- $S_8 = S_9 = \{r_1\}$
- $M_1 = M_3 = M_4 = M_8 = M_9 = \emptyset$
- $M_2 = M_{10} = \{r_1\}$
- $M_5 = M_6 = M_7 = \{r_2\}$

FIGURE 9. Tries $t_1$ and $t_2$ for rules $r_1$ and $r_2$ in Figure 1.

FIGURE 10. Trie $T$ obtained by merging $t_1$ and $t_2$ in Figure 9.
FUNCTION TrieSel(u, v, S)
/* u - Goal node, v - Trie node, S - set of selected rule-heads */
/* returns updated set of rule-heads */
BEGIN
1. if u is labeled by a variable RETURN(S)
2. \(M_1 = S \cap M_v\)
3. IF there is an edge \(e\) in \(T\) leaving \(v\) towards \(w\) having label of \(u\) THEN
4. BEGIN
5. \(S_1 = S \cap (S_w \cup M_w)\)
6. FOR each edge \(e_i\) leaving \(w\) towards a nonleaf node \(w_i\), DO
7. LET \(u_i\) be the \(i^{th}\) child of \(u\)
8. \(S_1 = S_1 \cap \text{TrieSel}(u_i, w_i, S_1)\)
9. END
10. ELSE \(S_1 = S \cap \overline{S_v}\)
11. RETURN(\(S_1 \cup M_1\))
END

FIGURE 11. Rule selection with trie \(T\).

call begins by inspecting the label of \(u\). Suppose \(u\) is labeled 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 labeled with a functor symbol and there is no edge leaving \(v\) that is labeled 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 the set complement of \(S_v\). Suppose there is an edge from \(v\) to \(w\) that has the same functor label. Then we descend to \(w\) in \(T\) because 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 \(S_w \cup M_w\). However, \(S_w \cup M_w\) may also have rules that are not in \(S\). Moreover, the root-to-leaf paths for some rules in \(S\) might have terminated at \(v\). Such rules must be present in \(M_v\). Therefore, we create two new sets \(S_1 = S \cap (S_w \cup M_w)\) 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 labeled 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 \(u_i, w_i\), and \(S_1\) 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. Formally, this is stated in the following theorem.

**Theorem 4.1.** TrieSel selects a rule iff it unifies (modulo nonlinearity) with the goal.

The proof details of this theorem appears in Appendix C.

Finally, we mention a modification to the method when the number of rules exceeds the number of bits in \(O(1)\) computer words. 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. 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, which is always less than or equal to the number of leaves in the rule.

5. IMPLEMENTATION AND PERFORMANCE

Our algorithms were not incorporated into any compiler and so their practical utility had not been established. We therefore began a project in June 1991 in collaboration with Applied Logic Systems (in Syracuse, New York) to seamlessly incorporate our indexing algorithms in their portable ALS compiler [2].

A preliminary implementation based on a naive and direct transformation of our algorithms into code showed that its usefulness was very limited. Only large and complex procedures (not often encountered in practice) seemed to benefit from them. However, typical Prolog programs have small procedures with shallow terms and few indexable arguments. Such programs did not benefit at all from the naive implementation. Even worse, it slowed down their execution speeds considerably.

The main problem with our preliminary implementation was that deep and shallow terms, terms with very few indexable arguments, and small and large procedures were all being handled uniformly. A serious drawback with such a uniform use of our algorithms is that indexing small procedures with shallow clause heads and few indexable arguments is expensive because it results in poor discrimination despite seeing many symbols. For such procedures, it is advantageous to use the simple method of first argument indexing. Although the latter indexing method can result in a lot of backtracking, the deterioration in overall execution speed is quite small when compared to using the former method. The problem now was how to realize the full benefits of our indexing technique (such as transparency, effectiveness, and reduced backtracking) over a broad spectrum of Prolog programs ranging from small procedures with a few shallow rules to complex procedures with deep structures, without unduly compromising the performance of any program in the spectrum.

Based on the above observations, it was evident that our implementation required mechanisms “sensitive” to term structures and sizes of procedures in order to beneficially extend its practical applicability. One approach to build-in such mechanisms is to do indexing in multiple stages; each stage further shrinks the size of the filtered set produced by the preceding stage using operations relatively more complex than the one used in earlier stages. The indexing process can be terminated at any stage, whenever it is not beneficial to continue further. So, small procedures with simple terms can be indexed quickly using the first few stages whereas all the stages are used for large and complex procedures. In the following text, we present an overview of our approach.

5.1. Multistage Indexing

The two main indexing methods—string-matching based and trie based—lend themselves quite nicely to decomposition into multiple stages. The operations performed
by these stages can gradually increase in complexity ranging from as simple an operation as first argument indexing done in the first stage to the complex operation of unification (modulo nonlinearity) performed in the last stage. To handle large and small procedures, we carefully interleave our string-matching based algorithm (beneficial for large procedures) with the trie based algorithm (useful for small procedures) during the indexing process.

In the current implementation of $\nu$-ALS, we have decomposed our indexing method into three stages. In the first stage, selection is done by examining the first argument symbol only. This is exactly the operation done by WAM indexing, which is also the indexing method in the ALS compiler. The second stage incorporates the improved selection described in Section 2.6. Specifically, this stage will select a set of clause heads such that the first string of every clause head in this set is either a prefix of the goal's first string or vice versa. In the third stage, we do unification (modulo nonlinearity) completely. Our strategy here is to use either the trie based algorithm or a combination of string-matching and trie based algorithms. We use the former method whenever the number of clause heads remaining after the previous stages does not exceed the size of a fixed number of words.

We mention that in the entire indexing process no symbol is ever examined more than once and hence no stage ever repeats the work done by any other stage. Furthermore, symbols in the goal subterms that occur within variable substitutions of every clause head are never examined.

5.1.1. CRITERION FOR CONTINUATION BEYOND A STAGE. Note that it should be possible to stop the indexing process at any stage whenever it is not beneficial to continue any further. We have a simple criterion for doing so in our current version. We stop the indexing process whenever the number of clause heads selected is 1. From both the first and second stages, we move on to the next stage whenever the $\text{try}_\text{me}-\text{retry}_\text{me}-\text{trust}_\text{me}$ chain selected has more than one clause. It is possible to develop a more sophisticated criterion by doing program analysis. For instance, we can analyze the clause heads to identify nonvariable positions in them. We can continue indexing if among the clause heads in the selected chain there are nonvariable symbols that have not been examined. Note that for any clause head in a selected $\text{try}_\text{me}-\text{retry}_\text{me}-\text{trust}_\text{me}$ chain we can identify at compile time exactly all of its symbols that would have been seen so far. Another possibility is to do mode analysis. If the remaining unscanned arguments of the goal are all outputs, then obviously there is no point continuing the indexing process any further.

5.1.2. INTERSTAGE INTERFACE. We now describe how the stages are hooked together. For this purpose we create three new WAM instructions: $\text{switch}_\text{on_automata}$, $\text{switch}_\text{on_string}$, and $\text{switch}_\text{on_trie}$. The $\text{switch}_\text{on_automata}$ instruction is used to begin the selection process of the second stage, whereas the latter two instructions are used for starting the third stage. The $\text{switch}_\text{on_string}$ instruction starts off our string-matching based algorithm, whereas the $\text{switch}_\text{on_trie}$ instruction initiates the trie based algorithm.

For describing the interface between the first and second stage we will assume familiarity with the internal organization of WAM appropriate for WAM indexing (see Section 5.9 on indexing in [7] for details). In WAM indexing, all clauses in a procedure are partitioned into four (not necessarily distinct) chains—$\text{variable}$, $\text{list}$, $\text{constant}$, and $\text{structure}$. The $\text{variable}$ chain merely links together all the clauses constituting a procedure. Every other chain contains a sequence of clauses that is
appropriate for that chain. For example, the list chain will contain clause heads whose first argument is a list.

An index block in WAM begins with the instruction switch_on_term followed by four pointers, each one pointing to one of the four chains. Note that the variable chain is used whenever the goal’s first argument is a variable (i.e., goal’s first string is empty). We can therefore regard its first string as being vacuously scanned and hence bypass the second stage altogether. So the pointer to the variable chain in the index block is replaced by a pointer to a memory block whose first instruction is either switch_on_string or switch_on_trie that when executed will initiate the third stage. For the list chain, the pointer is modified only when the number of clauses in it is more than one. In that case, the pointer is set to point to a memory block whose first instruction is a switch_on_automata that starts the second stage. For constant and structure chains, their corresponding pointers in the index block points to a table beginning with switch_on_constant and switch_on_structure instruction, respectively, followed by a sequence of entries. Each entry is a pair of the form \((k, @(single-clause))\) or \((k, @(multiple-clauses))\). The first component in a pair is the key. The pair \((k, @(single-clause))\) is entered into the table whenever there is only one clause in the procedure that matches the key in the pair, whereas the other one is used when there is more than one clause. In the former case, \(@(single-clause)\) points to the first instruction of the clause; in the latter case, \(@(multiple-clauses)\) points to the switch_on_automata instruction that starts the second stage. The switch_on_automata instruction is always followed by a parameter which is the state of the automaton from which the second stage will begin its transitions. We can thereby avoid rescanning the key symbol already seen in WAM indexing.

We now describe the interface to the third stage. Note that since any state of the automaton can be reached via the second stage, transition into the third stage can be done from any one of them. So we maintain a pointer final with each state. If the number of clauses in the try_me-retry_me-trust_me chain for a state is only one, then the final pointer of that state points to the starting address of the single clause. Otherwise, it points to a block of memory which starts either with a switch_on_trie instruction when the number of clauses is less than the word size or with a switch_on_string instruction otherwise.

Finally, we outline briefly how symbols seen in previous stages are not reexamined in the third stage. In the string-matching based algorithm, the problem is nonexistent since the same data structure (i.e., the Aho-Corasick automaton) is used uniformly in all the stages. The difficulty arises when combining the Aho-Corasick automaton with the trie used in the trie based algorithm. Observe that each state in the Aho-Corasick automaton corresponds to a position reached in the trie. So we maintain a pointer pos(s) in every state s to its corresponding position in the trie. Whenever the trie based algorithm is initiated from a state, say \(\alpha\) at run time, then we traverse the trie from \(pos(\alpha)\) onward, thereby avoiding reexamination of symbols seen prior to reaching \(pos(\alpha)\).

Finally, we remark that it is possible to introduce additional stages between the intermediate and final stage. For example, one such stage can be designed to do the same operation as that of the intermediate stage, but using the last argument instead. Our main objective in this first prototype of \(\nu\)-ALS was to validate the practical applicability of our indexing method. For ease and simplicity of implementation we chose to design only one intermediate stage.
### Table 2. Speedup Figures

<table>
<thead>
<tr>
<th>Program</th>
<th>Query</th>
<th>Loops</th>
<th>ALS</th>
<th>ν-ALS</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>const</code></td>
<td><code>c(f(100))</code></td>
<td>50,000</td>
<td>168.76</td>
<td>9.48</td>
<td>17.74</td>
</tr>
<tr>
<td></td>
<td><code>c(f(50))</code></td>
<td>50,000</td>
<td>90.37</td>
<td>8.83</td>
<td>10.23</td>
</tr>
<tr>
<td></td>
<td><code>c(f(1))</code></td>
<td>50,000</td>
<td>7.97</td>
<td>8.68</td>
<td>0.92</td>
</tr>
<tr>
<td><code>full16</code></td>
<td><code>f(1, 3, X, 32)</code></td>
<td>50,000</td>
<td>62.12</td>
<td>21.15</td>
<td>2.94</td>
</tr>
<tr>
<td><code>border</code></td>
<td><code>borders(yugoslavia, mediterranean)</code></td>
<td>50,000</td>
<td>16.03</td>
<td>8.15</td>
<td>1.97</td>
</tr>
<tr>
<td></td>
<td><code>borders(yugoslavia, hungary)</code></td>
<td>50,000</td>
<td>12.68</td>
<td>8.48</td>
<td>1.50</td>
</tr>
<tr>
<td></td>
<td><code>borders(yugoslavia, albania)</code></td>
<td>50,000</td>
<td>7.12</td>
<td>8.48</td>
<td>0.84</td>
</tr>
<tr>
<td><code>dnf</code></td>
<td><code>s1</code></td>
<td>5,000</td>
<td>93.17</td>
<td>68.48</td>
<td>1.36</td>
</tr>
<tr>
<td></td>
<td><code>s2</code></td>
<td>5,000</td>
<td>43.98</td>
<td>34.65</td>
<td>1.27</td>
</tr>
<tr>
<td></td>
<td><code>s3</code></td>
<td>5,000</td>
<td>22.95</td>
<td>19.48</td>
<td>1.18</td>
</tr>
<tr>
<td><code>replace</code></td>
<td><code>replace(neg(expr), A, B)</code></td>
<td>50,000</td>
<td>9.92</td>
<td>9.82</td>
<td>1.01</td>
</tr>
<tr>
<td></td>
<td><code>replace((2 &lt; 4), A, B)</code></td>
<td>50,000</td>
<td>10.72</td>
<td>10.82</td>
<td>0.99</td>
</tr>
<tr>
<td><code>replace.sw</code></td>
<td><code>replace(1, A, B, neg(expr))</code></td>
<td>50,000</td>
<td>69.08</td>
<td>21.43</td>
<td>3.22</td>
</tr>
<tr>
<td></td>
<td><code>replace(1, A, B, (2 &lt; 4))</code></td>
<td>50,000</td>
<td>48.18</td>
<td>22.43</td>
<td>2.15</td>
</tr>
<tr>
<td></td>
<td><code>replace(1, A, B, mul(e1, e2))</code></td>
<td>50,000</td>
<td>15.90</td>
<td>22.50</td>
<td>0.71</td>
</tr>
<tr>
<td><code>replace.all</code></td>
<td><code>replace(1, A, B, neg(expr))</code></td>
<td>50,000</td>
<td>68.60</td>
<td>21.37</td>
<td>3.21</td>
</tr>
<tr>
<td></td>
<td><code>replace(1, A, B, (2 &lt; 4))</code></td>
<td>50,000</td>
<td>69.70</td>
<td>22.45</td>
<td>3.10</td>
</tr>
<tr>
<td></td>
<td><code>replace(1, A, B, mul(e1, e2))</code></td>
<td>50,000</td>
<td>73.33</td>
<td>21.57</td>
<td>3.40</td>
</tr>
<tr>
<td><code>ll</code></td>
<td><code>p</code></td>
<td>10,000</td>
<td>62.98</td>
<td>56.68</td>
<td>1.11</td>
</tr>
<tr>
<td><code>ll(2)</code></td>
<td><code>q</code></td>
<td>10,000</td>
<td>87.32</td>
<td>70.80</td>
<td>1.23</td>
</tr>
<tr>
<td><code>ll(3)</code></td>
<td><code>r</code></td>
<td>10,000</td>
<td>59.50</td>
<td>45.37</td>
<td>1.31</td>
</tr>
<tr>
<td><code>queens</code></td>
<td><code>get_solutions(A)</code></td>
<td>20</td>
<td>49.28</td>
<td>49.68</td>
<td>0.99</td>
</tr>
</tbody>
</table>

### 5.2. Experimental Results

Table 2 lists timing results of the benchmark programs we have tested. Each program is tested on one or more queries. The figures under Loops indicate the number of times the corresponding query is run. Times are measured in seconds and speedups are computed as the ratio of the running time of programs compiled using ALS over that using ν-ALS. All programs are run on a Sun-3/160. The programs include the Dutch national flag (Figure 12) problem given in [12], the border (Figure 12) predicate in the CHAT-80 system [20], the replace (Figure 14) program and the 8-queens (Figure 13) problem, both of which are ALS benchmarks, and programs to parse LL(k) (Figure 15) grammars. The first two programs in Table 2, viz. `const` and `full16` (both in Figure 12) are atypical programs contrived to highlight key aspects of our indexing algorithm.

The `const` program illustrates the advantage of ν-ALS whenever it is beneficial to look beyond the first argument. On the query `c(f(100))`, WAM indexing stops after inspecting the principal functor `f` of the first argument. However, nothing is achieved at all since all the 100 clauses have the same symbol there. Therefore, it has to resort to unification and backtracking through 99 clauses before the right one is finally found, while the intermediate stage in our method will inspect one more symbol and find the right answer without doing any unification. The response to queries `c(f(1))` and `c(f(50))` are essentially the same except that backtracking is done only 1 and 50 times, respectively, in WAM indexing.

The `border` program illustrates the performance of ν-ALS on facts, especially ground facts wherein the arguments are all constant terms. Since the arguments are
**FIGURE 12.** Benchmark programs \textit{const, full16, border,} and \textit{dnf}.

**FIGURE 13.** Benchmark program \textit{queens}.
all constant terms, we can pick the correct fact at the end of the second stage. Note that the lower the priority of the selected fact, the bigger the speedup (compare the speedups on `-border(yugoslavia, mediterranean)` vs. `-border(yugoslavia, albania)`). This can also be seen on the `const` program (see speedups on `-c(f(1))` vs. `-c(f(50))`).

Our method can naturally exploit the extant practice of writing programs with input arguments preceding all output arguments. We can speed up such programs, especially in those cases in which input arguments that are nonvariables precede those that have variables as arguments. For example, consider the program for the Dutch national flag problem. With the exception of the first clause, every other clause’s first argument is a list whose first element is a constant. WAM indexing will look through the clauses one by one while our intermediate stage finds the right clause based on a single transition.

Our indexing method has the potential to reduce nondeterminism in Prolog programs that have many nonvariable symbols in clause heads; the larger the number of such symbols, the better is the speedup. For example, by putting the lookahead symbols in the clause heads, we can parse `LL(k)` grammars deterministically, that is, our indexing method always selects only one clause head. Therefore, no choice points are ever created, thereby resulting in deterministic execution. Observe that the speedups increase with larger k.

In all of the examples discussed above, we only needed to use the first two stages of our indexing algorithm. We now discuss the impact of the third stage. We use program `replace` (an ALS benchmark) for illustration. In this program, WAM indexing suffices to pick the right clause in all the queries shown in the table. Hence our speeds are comparable to that of ALS. So in `replace.sw`, we switch the first argument of every clause in `replace` to the last and put a constant 1 at its original position in order to nullify the effect of the first two stages. Therefore, all the 12 clauses appear as input to the final stage. The queries that look for the last and fourth clause in the program obtain speedups of 3.22 and 2.15, respectively. There is a slowdown when selecting the first clause. This is because the overhead in our final stage becomes a dominant factor when selecting very high priority clauses. In our current implementation, we have observed that we start gaining whenever we have to select clauses with priority greater than three. Program `full16` is another example that uses all three stages.

Observe that we always gain when handling queries that need all answers. See the results for the program `replace.all`. This program is identical to `replace.sw`, the only difference being that iterations are now using `loop2` (see Figure 13) that does not use a cut. Therefore, each iteration finds all answers to the query. All other programs use `loop1` (see Fig. 13) that uses a cut to stop the search upon finding the first correct answer. Finally, the 8-queens problem in the ALS benchmarks is representative of programs that do not benefit from our indexing. Our results show that the run-time performance of such programs is not affected by our technique.

5.3. Discussion

Our algorithms and the implementation can be enhanced in several ways. In the following text, we discuss some of them. Note that in our methods the terms are scanned in preorder to perform selection. However, they can be easily adapted to use any depth-first traversal. Specifically, we can choose a traversal that inspects
FIGURE 14. Benchmark programs replace and replace.sw.

![Benchmark programs replace and replace.sw.](image)

FIGURE 15. Benchmark programs for $ll(k)$ grammars.
arguments other than the first one in the clause heads. This is especially useful for indexing rules in which variable arguments precede nonvariables. Since the nonvariable symbols in the clause heads are now seen early, we may be able to complete indexing in fewer stages. For example, while indexing terms such as those given in the \textit{replace.sw} program (see Figure 14), the indexing can be completed in the second stage itself. In contrast, using preorder the indexing proceeds to the third stage also. In addition, the flexibility in choosing a traversal order allows us to use other global analysis techniques such as mode analysis to speed up indexing. For example, using mode analysis, we can synthesize a traversal that will compare the nonvariable parts of the goal and clause heads as early as possible, thereby reduce indexing time.

Some Prolog compilers index over multiple arguments. Our indexing method may have to use all three stages to inspect such arguments. This may not always be efficient. However, it is possible to extend our method for handling multiple arguments efficiently. Specifically, the clause heads can be decomposed into a forest of their arguments. We then index on each (or a selected set) of these arguments separately. Note that, while doing so, we can choose different traversal orders for each of these arguments to further speedup indexing.

There is yet another important optimization step, and this has to do with doing unification following indexing. If, during the indexing step, we compute at most one substitution for any variable, then there is no need to do unification. In case we do have to perform unification, then we must avoid rescanning symbols already seen during indexing. Incorporating these optimizations will further improve the performance of programs. It is quite likely that even such queries as \textit{c(f(1)), borders(yugoslavia, albania)}, and \textit{replace(1, A, B, multiple(expr1, expr2))} may not exhibit any slowdown.

Observe that our implementation has been fine tuned for improving the performance of Prolog programs with shallow heads and few indexable arguments. This approach is not suitable for programs that have “potentially useful” indexing information embedded within the program body. In order to realize the full benefits of our indexing methods on such programs, such information will have to be propagated into the clause heads. We have recently developed such a technique [6]. Application of this technique results in large and complex heads which can then be exploited in our indexing methods. For example, \textit{ll(2)} and \textit{ll(3)} programs given in Figure 15 are the result of applying techniques in [6] to Prolog programs that correspond to direct translation of the two grammars.

6. CONCLUSION

In this paper we described two new techniques based on unification (modulo non-linearity) for indexing Prolog programs. These techniques appeared to be beneficial to atypical Prolog programs (e.g., deep term structures and large number of clauses constituting procedures). To extend its applicability to a broad spectrum of Prolog programs, we decomposed the technique to do indexing in multiple stages. Each stage further shrinks the size of its input set by employing operations relatively more complex than those used in previous stages. We validated the practical viability of our approach by showing good speedups over a range of Prolog programs typically encountered in practice.
Both our indexing methods may examine symbols not relevant for indexing. In fact, no indexing method based on depth-first traversal can always avoid looking at unnecessary symbols. Design of optimal indexing algorithms based on adaptive traversals that examine only symbols relevant for indexing is an interesting and important open problem.

APPENDIX A. CORRECTNESS AND COMPLEXITY OF PROCEDURE INDEX

We now establish correctness of procedure \textit{Index}. Specifically, we show that \( r \) unifies (modulo nonlinearity) with \( g \) iff procedure \textit{Index} terminates with \( \text{fail} = \text{false} \). First we need the following definitions to set up the formal machinery for proving correctness.

\textit{Definition A.1.} The preorder string of a term \( t \) is the string obtained by replacing every node in the preorder sequence of \( t \) by its label.

\textit{Definition A.2.} (Position). Position of a node \( v \) in a term, denoted \( \text{pos}(v) \) is the empty string \( \Lambda \) if \( v \) is the root or \( p.i \) if \( v \) is the \( i \)th child of its parent whose position is \( p \).

For example, for node 5 in Figure 1, \( \text{pos}(5) \) is 1.2.1.

To establish correctness we need the following properties of term trees. Let \( c_1 \) and \( c_2 \) denote the preorder traversal sequences of \( t_1 \) and \( t_2 \), respectively. Let \( c_1(i) \) and \( c_2(i) \) denote the \( i \)th node in \( c_1 \) and \( c_2 \), respectively, and \( s_1 \) and \( s_2 \) denote the preorder strings obtained from \( c_1 \) and \( c_2 \), respectively.

\textit{Lemma A.1.} If \( s_1 = s_2 \), then \( \text{pos}(c_1(i)) = \text{pos}(c_2(i)) \) (\( 1 \leq i \leq |c_1| = |c_2| \)).

\textbf{Proof.} By induction on \( i \).

\textbf{Base case:} \( i = 1 \). Since \( c_1(i) \) and \( c_2(i) \) are both root nodes, \( \text{pos}(c_1(i)) = \Lambda = \text{pos}(c_2(i)) \).

\textbf{Induction step:} Assume that the lemma is true for all \( i < k \). Let \( u_1 \) and \( u_2 \) be the nodes at \( c_1(k) \) and \( c_1(k + 1) \), respectively. Similarly, let \( v_1 \) and \( v_2 \) be nodes at \( c_2(k) \) and \( c_2(k + 1) \), respectively.

\textbf{Case 1.} \( u_1 \) is not a leaf node. Since the arity of a functor symbol is unique and both \( u_1 \) and \( v_1 \) have the same label (as \( s_1 = s_2 \)), \( u_1 \) is not a leaf node also. By induction hypothesis \( \text{pos}(u_1) = \text{pos}(v_2) \). Since \( c_1 \) and \( c_2 \) are preorder sequences, \( u_2 \) and \( v_2 \) must be the first children of \( u_1 \) and \( v_1 \), respectively. Therefore, \( \text{pos}(v_1) = \text{pos}(v_2) \).

\textbf{Case 2.} \( u_1 \) is a leaf node. So is \( v_1 \) because functors have unique arity. Let \( u_1 \) be the last node (in preorder) in the subtree rooted at ancestor \( x_i \) of \( u_1 \). Similarly, let \( v_1 \) be the last node in the subtree rooted at ancestor \( y_j \) of \( v_1 \). We claim that \( \text{pos}(x_i) = \text{pos}(y_j) \). Suppose this is not true. Since \( \text{pos}(u_1) = \text{pos}(v_1) \), \( u_1 \) and \( v_1 \) must have the same number of ancestors. Therefore, there must be an ancestor \( x_j \) of \( u_1 \) such that \( \text{pos}(x_j) = \text{pos}(y_j) \). Assume without loss of generality that \( x_j \) is a descendant of \( x_i \). This
situation is illustrated in Figure 16. Now \( y_{j-1} \) must have at least one uninspected child and this must be \( v_2 \) (because \( v_2 \) follows \( v_1 \) in preorder). Let \( y_j \) be the \( m \)th child of \( y_{j-1} \). Since \( pos(x_j) = pos(y_j) \), it follows that \( x_j \) is also the \( m \)th child of \( x_{j-1} \). Clearly, all children of \( x_{j-1} \) must have been visited (see Figure 16). Therefore, the arity of the label on \( x_{j-1} \) must be \( m \). Now note that the arity of \( y_{j-1} \) must be at least \( m + 1 \). Therefore, \( x_{j-1} \) and \( y_{j-1} \) cannot have the same label. Note also that the preorder number of both \( x_{j-1} \) and \( y_{j-1} \) must be \( \leq k \). Let \( c_1(n) \) be \( x_{j-1} \). By induction hypothesis, \( pos(c_2(n)) = pos(x_{j-1}) = pos(y_{j-1}) \). Therefore, \( c_2(n) = y_{j-1} \), because there cannot be more than one node at a given position in a term. Hence, \( c_1(n) \) and \( c_2(n) \) have different labels and so \( s_1 \neq s_2 \) — a contradiction. Therefore, \( pos(x_i) = pos(y_j) \).

Let \( pre_1 \) and \( pre_2 \) denote prefixes of length \( l \) of preorder strings \( s_1 \) and \( s_2 \), respectively. Let \( u_1 \) and \( u_2 \) be the nodes in \( c_1(l + 1) \) and \( c_2(l + 1) \), respectively.

**Corollary A.1.** If \( pre_1 = pre_2 \), then \( pos(u_1) = pos(u_2) \).

**Proof.** Observe that the proof of the induction step in the above lemma does not use the labels on the nodes in \( c_1(l + 1) \) and \( c_2(l + 1) \). Hence the same proof can be directly applied here. ☐

**Lemma A.2.** \( t_1 = t_2 \) iff \( s_1 = s_2 \).

**Proof.** \( \Rightarrow \) Obvious.

\( \Leftarrow \) If \( s_1 = s_2 \), then \( pos(c_1(i)) = pos(c_2(i)) \) (\( 1 \leq i \leq |c_1| = |c_2| \)), i.e., for every node \( u(v) \) in \( t_1 \) (\( t_2 \)) there is a node \( v(u) \) in \( t_2 \) (\( t_1 \)) such that \( pos(u) = pos(v) \) and labels on \( u \) and \( v \) are identical. This means \( t_1 = t_2 \). ☐

Let \( R[i, j] \) denote the subarray of \( R \) between \( i \) and \( j \) (excluding the element \( R[j] \)). Let \( string(R[i, j]) \) represent the string obtained by concatenating the label
fields of $R[i, j]$. Let $E_R[1, p_r]$ be the string obtained by replacing each variable $x$ in $\text{string}(R[1, p_r])$ by the $\text{string}(G[u, v])$, where the subarray $G[u, v - 1]$ contains information about the subtree computed as substitution for $x$ by procedure $\text{Index}$. Similarly, let $E_G[1, p_g]$ denote the string obtained by a similar process on $G[1, p_g]$. Based on the above properties, we can now assert the following lemma.

**Lemma A.3.** At the end of an iteration, if $\text{fail} = \text{false}$, then $E_G[1, p_g] = E_R[1, p_r]$.

**Proof.** By induction on number of iterations.

**Base case:** $n = 0$ (i.e., prior to entry of loop). We must have executed either lines 7–8 or 10–11 before entering the while loop. In both cases, the lemma holds if $\text{fail} = \text{false}$ (by Theorems 2.1 and 2.2).

**Induction step:** Assume that the lemma holds at the end of the $k$th iteration. The $(k + 1)$th iteration performs one or more iterations of loop at line 13 followed by execution of lines either between 30–34, 36–40, 44–49, or 51–56. We first show that at the end of each iteration of the loop at line 13 the lemma holds if it holds at the beginning of this iteration. Assume that in the loop the lines 15–17 are executed. By the definition of $E_G[1, p_g]$ and $E_R[1, p_r]$, it follows that after execution of the lines 15–17, $E_G[1, p_g] = E_R[1, p_r]$. The same is true if the lines 19–21 are executed instead of lines 15–17. Therefore, at the beginning of line 24 of the $(k + 1)$th iteration, $E_G[1, p_g] = E_R[1, p_r]$. If $\text{fail} = \text{false}$ after executing lines 30–34 or 36–40, then by Theorem 2.1 we append equal strings to $E_G[1, p_g]$ and $E_R[1, p_r]$ and hence the lemma holds at the end of this iteration. On the other hand, if lines 44–49 or 51–56 are executed, then by Theorem 2.2 equal strings are again appended to $E_G[1, p_g]$ and $E_R[1, p_r]$ and so the lemma holds at the end of this iteration.

Finally we can now establish the correctness theorem.

**Theorem A.1 (Correctness).** $r$ unifies with goal (modulo nonlinearity) iff procedure $\text{Index}$ terminates with $\text{fail} = \text{false}$.

**Proof.** $\Leftarrow$ When procedure $\text{Index}$ terminates with $\text{fail} = \text{false}$, then both $p_g$ and $p_r$ point to last elements in $G$ and $R$, respectively. At this point $E_G[1, p_g]$ and $E_R[1, p_r]$ are identical by Lemma A.3. Let $t_1$ denote the term whose preorder string is $E_G[1, p_g]$ and let $t_2$ denote the term whose preorder string is $E_R[1, p_r]$. Now by Lemma A.2, $t_1 = t_2$. Procedure $\text{Index}$ constructs $E_G$ and $E_R$ by replacing some variable occurrences in $g$ and $r$ by preorder strings of terms. Therefore, $r$ unifies (modulo nonlinearity) with $g$.

$\Rightarrow$ Suppose $\text{fail} = \text{true}$ upon termination of procedure $\text{Index}$. The only way for $\text{fail}$ to become true is when one of the string-matching operations does not succeed. Just before this string-matching operation, $E_G[1, p_g] = E_R[1, p_r]$ (by proof of Lemma A.3). Assume that this string match failed because $G[p_g + x].\text{label} \neq R[p_r + x].\text{label}$. Now note that $E_G[1, p_g + x - 1] = E_R[1, p_r + x - 1]$. By Corollary A.1 it follows that the node at $\text{pos}(G[p_g + x].\text{node}) = \text{pos}(R[p_r + x].\text{node})$. This means there are two nodes $u$ in $g$ and $v$ in $r$ such that $\text{pos}(u) = \text{pos}(v)$ whose labels do not match. Therefore, $g$ cannot unify (modulo nonlinearity) with $r$. \qed
We now establish worst-case time complexity of procedure Index. Let $l_g$ and $k_g$ be the number of leaves and number of variable occurrences in the goal. Similarly, let $l$ and $k$ denote the number of leaves and number of variable occurrences in rule $r$. Then we can state the following theorem.

**Theorem A.2.** The worst-case time required by procedure Index to select rule head $r$ is $O(\min\{k + k_g, l, l_g\})$.

**Proof.** Note that each iteration of the inner loop (i.e., lines 13–23) computes exactly one substitution which takes $O(1)$ time. For every iteration of the outer loop, the inner loop is executed at least once. Note that the other steps in the outer loop (lines 24–59) take only $O(1)$ time because they perform at most one string match operation. Therefore, the worst-case time complexity of the outer loop cannot be more than the worst-case time complexity of the inner loop. The inner loop requires in the worst-case, time proportional to the number of substitutions computed and this number is $\leq O(k + k_g)$.

Let $x$ be the number of variables in goal that took substitutions. The remaining $l_g - x$ leaves must occur in subterms of the goal that are computed as substitutions for rule variables. Clearly, each such substitution must have at least one goal leaf in it. Therefore, the total number of substitutions computed for rule variables must be $\leq O(l_g - x)$. Therefore, the total number of substitutions for both rule and goal variables is $\leq O(l_g)$. By a similar argument, the total number of substitutions is also $\leq O(l)$.

Finally all the steps preceding the outer loop in procedure Index (lines 1–11) require $O(1)$. Therefore, the worst-case time complexity of procedure Index is $O(\min\{k + k_g, l, l_g\})$. □

An important feature of procedure Index is that the goal needs to be scanned only once regardless of the number of rules to be selected. Suppose $r_1, r_2, \ldots, r_m$ are the rules to be selected. Let $l_i$ and $k_i$ denote the number of leaves and the number of variable occurrences in $r_i$. Then we can state the following corollary.

**Corollary A.2.** The worst-case time complexity for selecting the $m$ rules is $O(|g| + \sum_{i=1}^{m}\min\{k_g + k_i, l_g, l_i\})$.

**Proof.** $G$ used in procedure Index can be constructed from $g$ in $|g|$ time. Then procedure Index is invoked once to select each $r_i$. Hence the result. □

**Appendix B. Details of the Demand-Driven Algorithm**

We now present the details of the demand-driven indexing algorithm. The algorithm is implemented using three procedures: Index, DemandScan, and Select. Procedure Index is obtained by modifying procedure Index to deal with scanning the goal on demand. Specifically, these modifications mainly deal with the steps in Index that compute substitutions for rule variables and perform string matching. (These modifications in Index are annotated with comments in boldface.) Observe that the substitutions for rule variables can suspend an instance of Index if the subtree skipped has not been scanned by the demand-driven algorithm. Also
observe that some string-matching operations are done in two stages. In the first stage, a string match is performed in $O(1)$ time using the currently scanned portion of the goal. In the second stage it is done by scanning goal symbols one at a time on demand and then comparing them with rule symbols. In the algorithm, a boolean function $\text{defined}$ is used to verify whether a field in $G[i]$ is initialized.

**Procedure** $\text{Index}_i$

BEGIN

01. $\text{fail}[i] := \text{FALSE}$;
02. { Perform first string match operation}
03. $p_g[i] := p_r[i] := 1$;
04. IF $\text{defined}(G[p_g[i]].\text{varposn})$ THEN
05.      $l_g[i] := G[p_g[i]].\text{varposn}$ \{$l_g[i]$ is the length of goal string \}
06. ELSE $l_g[i] := \text{end}_g$;
07. ENDIF;
08. $l_r[i] := R[i, p_r[i]].\text{varposn}$ \{$l_r[i]$ is the length of rule string \}
09. IF $l_g[i] > l_r[i]$ THEN
10.      \{ goal string is longer \}
11.      $\text{fail}[i] := G[l_g[i]].\text{state} \neq R[i, l_r[i]].\text{state}$;
12. ELSE
13.      \{ rule string is longer than currently scanned portion of goal string. So, perform string matches in two stages. First stage uses string matching operation using automaton states. \}
14.      $\text{fail}[i] := (G[l_g[i]].\text{state} \neq R[i, l_r[i]].\text{state})$;
15.      $p_r[i] := p_g[i] := l_g[i] + 1$;
16. \{ The second stage gets one goal symbol at a time on demand and does symbol-by-symbol comparison \}
17. \begin{itemize}
18. \item WHILE $\text{fail}[i] = \text{TRUE}$ \& (G and R are not completely scanned) \& (both $R[i, p_r[i]].\text{label}$ and $G[i, p_g[i]].\text{label}$ are functors) DO
19. \item $v_g := \text{TRUE}$;
20. \item $p_g[i] := p_g[i] + 1$;
21. \item $p_r[i] := R[i, p_r[i]].\text{subtree} + 1$;
22. ENDIF;
23. \item WHILE $\text{fail}[i] = \text{TRUE}$ \& (G and R are not completely scanned) DO
24. \item WHILE $G[p_g[i]].\text{label}$ \& $R[i, p_r[i]].\text{label}$ is a variable DO
25. \item IF $G[p_g[i]].\text{label}$ is a variable THEN
26. \begin{itemize}
27. \item compute substitution for goal variable \}
28. \item $v_g := \text{TRUE}$;
29. \item $p_g[i] := p_g[i] + 1$;
30. \item $p_r[i] := R[i, p_r[i]].\text{subtree} + 1$;
31. END; \{ while \}
32. END; \{ while \}
33. \item ENDIF;
34. \item END; \{ while \}
35. \end{itemize}
36. \end{itemize}
va := FALSE;
p_{r[i]} := p_{r[i]} + 1;
{check if this skipped subtree is already completely
scanned. If not suspend and await completion of
scanning.}

IF ~defined(G[p_{pg[i]}.subtree]) THEN
suspend[i] := TRUE;
suspend;
{upon restart we will continue from here. Also scan-
ing of skipped subtree is complete when we are
restarted}

ENDIF;
p_{g[i]} := G[p_{g[i]}.subtree] + 1;
ENDIF;
END;

IF G and R are not completely scanned THEN
{Both p_{g[i]} and p_{r[i]} point to functor nodes and v_g
specifies whether the immediately preceding sub-
stitution is for the rule or goal variable. As this is
demand-driven scanning, check whether there are
enough symbols available in the goal.}

IF defined(G[p_{pg[i]}.varposn]) THEN
l_{g[i]} := G[p_{pg[i]}.varposn] - p_{g[i]} + 1
ELSE l_{g[i]} := end_{g} - p_{g[i]} + 1;
ENDIF;
l_{r[i]} := R[i, p_{r[i]}.va, 'posn - p_{r[i]} + 1;
IF ~v_g THEN
{Last substitution is for rule variable. So check for occur-
rence of prefix of current rule string in current goal string}

IF l_{g[i]} > l_{r[i]} THEN
{Check occurrence of entire rule string in goal string as in
Figure 3a}

pre_{g} := pf(G[p_{g[i]} + l_{r[i]} - 1].state);
pre_{r} := pf(R[i, p_{r[i]} + l_{r[i]} - 1].state);
nd_{r} := nd(R[i, p_{r[i]} + l_{r[i]} - 1].state);
fail[i] := ~(pre_{r} ≤ pre_{g} ≤ pre_{r} + nd_{r});
p_{r[i]} := p_{r[i]} + l_{r[i]}; p_{g[i]} := p_{g[i]} + l_{r[i]};
ELSE
{Rule string is longer than currently scanned por-
tion of the goal string. So perform string matches
in two stages. First, check for occurrence of prefix
of rule string (of length l_{g[i]}) in goal string as in
Figure 3b.}
preg := pf(G[pg[i] + lg[i] - 1].state);
prer := pf(R[i, pr[i] + lg[i] - 1].state);
nd := nd(R[i, pr[i] + lg[i] - 1].state);
fail[i] := \neg(pre_r \leq preg \leq pre_r + nd);
pr[i] := pr[i] + lg[i];
pq[i] := pq[i] + lg[i];
pg[i] := pq[i] + 1; pr[i] := pr[i] + 1;
IF fail[i] THEN return;
ENDIF;

WHILE fail[i] = FALSE \land (G and R are not completely scanned) \land (both R[i, pr[i]].label and G[i, pq[i]].label are functors) DO
  resume DemandScan;
  fail[i] := G[end].label = R[i, pr[i]].label;
  pq[i] := pq[i] + 1; pr[i] := pr[i] + 1;
END; { while }
ELSE {Last substitution is for goal variable, so check occurrence of prefix of current goal string in current rule string }

IF lg[i] < lr[i] THEN
  {Again rule string is larger than the currently scanned goal string. So do string matching in two stages. First check occurrence of currently scanned prefix of goal string in rule string as in Figure 3c }

preg := pf(G[pg[i] + lg[i] - 1].state);
pre_r := pf(R[i, pr[i] + lg[i] - 1].state);
nd_g := nd(G[pg[i] + lg[i] - 1].state);
d_g := depth(G[pg[i] + lg[i] - 1].state);
fail[i] := \neg(preg \leq pre_r \leq preg + nd_g) \lor \neg(l_g[i] = d_g)
pr[i] := pr[i] + lg[i];
pq[i] := pq[i] + 1 + lg[i];
IF fail[i] THEN return;
ENDIF;

WHILE fail[i] = FALSE \land (G and R are not completely scanned) \land (both R[i, pr[i]].label and G[i, pq[i]].label are functors) DO
  resume DemandScan;
  fail[i] := G[end].label = R[i, pr[i]].label;
  pq[i] := pq[i] + 1; pr[i] := pr[i] + 1;
END; { while }
ELSE {Check occurrence of proper prefix of goal string in rule string as in Figure 3d }

preg := pf(G[pg[i] + lr[i] - 1].state);
Procedure DemandScan constructs the goal array $G$ incrementally by traversing the goal tree recursively in preorder. It scans the goal nodes one at a time, suspending itself after visiting each node. It is resumed either when the next node is needed by Index$_i$ (for some $i$) or when all the rules are suspended. In the latter case, abort is true and DemandScan enters a skip phase. In this phase it returns without initiating any new recursive calls. Specifically, suppose $v$ is the last node visited and $u$ is its closest ancestor that has a rule $r_i$ suspended at its leftmost uninspected child. DemandScan will return without initiating new recursive calls at any of the ancestors of $v$ below $u$. The subtrees skipped in this process are not needed for indexing and can now be regarded as having been scanned. On returning to $u$, DemandScan resumes normal operation by making $r_i$ active.

**Procedure** DemandScan($v : node$);

VAR $x : integer$;

BEGIN
01. $G[end_g + 1].label := v.label$;
02. $x := end_g := end_g + 1$
03. move any rule suspended on $v$ to active queue;
04. IF $v.label$ is a variable THEN
   { this node of the goal tree is labeled by variable and hence terminates the current goal string. So update varposn field of records in $G$ that belongs to this goal string. }
05. FOR $i := end_g$ DOWN TO 1 DO
06. IF defined($G[i].varposn$) THEN exit loop;
07. ELSE $G[i].varposn := x$;
08. ENDIF;
09. END;
   {As the goal node is a variable it is now processed completely. So suspend and await next request. }
10. suspend;
11. return
12. ENDIF;
   {For nonvariable nodes we need to scan the functor with Aho-Corasick automaton. ACScan performs scanning with the automaton. }
13. IF ($x = 1$) OR defined($G[x - 1].varposn$) THEN
   {First find out whether this is the first symbol of a new string. If it is scan from the startstate of the automaton }
G[x].state := ACScan(startstate, v.label)

ELSE

{ Otherwise continue scanning this symbol from where we left. }

G[x].state := ACScan(G[x - 1].state, v.label);

ENDIF

{ Next fill the fringenode field so that a rule that skips the subtree rooted at v can be suspended appropriately. If the fringe stack is empty then by skipping the subtree rooted at v scanning of goal is complete. So, G[x].fringenode is left undefined. }

IF !empty(fringestack) THEN

G[x].fringenode := Top(fringestack);

ENDIF

{ We are done with processing this node. So suspend and await request to scan further. }

suspend

IF (v has a child) THEN

{ v is visited and has un inspected children and therefore is a fringe node. Push it onto the fringe stack }

push(v, fringe stack);

{ Are we in skip phase? Check abort to find out }

IF !abort THEN

{ abort is false. We are restarted by Index; that now needs the next goal node. So, initiate recursive calls at children of v. }

FOR i := 1 TO arity(v.label) DO

IF (i = arity(v.label)) THEN

{ If this is the last child of v then v is no longer a fringe node. So pop it of the fringe stack }

pop(fringestack);

ENDIF

ENDIF

DemandScan(ith child of v);

{ we may return from recursive call in skip phase so perform the following check to see if we need to skip scanning subtrees rooted at remaining children of v }

IF i < arity(v.label) \&\& abort THEN

IF (i + 1)th child has no rule suspended on it THEN

pop(fringestack)

exit loop

ELSE abort := FALSE;

ENDIF;

ENDIF;

END

ELSE

{ abort is true. So we are in skip mode. }

pop(fringestack)
41. ENDIF;
42. ENDIF;
   { finally before returning from the recursive call update the
   subtree field to indicate that the scanning of subtree rooted
   at \( v \) is complete }
43. \( G[x].\text{subtree} := \text{end}_g \)
END;{ DemandScan }

The indexing algorithm begins by invoking procedure \textit{Select}. This procedure coordinates \textit{Index} and \textit{DemandScan}. It begins by invoking \textit{DemandScan} at the root of the tree to start the construction of \( G \). After \textit{DemandScan} visits root, it suspends itself and returns control to \textit{Select} again. Following this, \textit{Select} creates active and suspend queues and places all the rules in the active queue. It then picks a rule from active queue and starts \textit{Index}. When \textit{Index} returns, \textit{Select} checks whether the selection of \( r_i \) has failed, succeeded, or suspended. In case of failure or successful completion, \textit{Index} is terminated. Otherwise \textit{Index} is stopped and \( r_i \) is moved to the suspend queue. (Note that the suspended rules are moved into active queue by \textit{DemandScan} after the subtrees skipped by them are scanned completely.) Following this, \textit{Select} picks the next rule from the active queue and continues the selection of this rule. Suppose there are no active rules. Then \textit{Select} checks the suspend queue. If it is empty, then the indexing algorithm terminates. Suppose suspend queue is not empty. Then subtrees of goal need to be skipped. Therefore, \textit{Select} instructs \textit{DemandScan} to enter skip phase by setting \textit{abort} to true and restarting it. Upon return from \textit{DemandScan}, at least one rule must be moved from suspend queue to active queue. Therefore, \textit{Select} resumes its normal operation of picking an active rule and continuing with its selection.

\textbf{Procedure} \textit{Select};
BEGIN
01. \textit{end}_g := 0;
02. \textit{DemandScan}(root);
03. Place all rules in active queue.
04. WHILE \neg\textit{empty}(active queue) \lor\neg\textit{empty}(suspend queue) DO
05.   WHILE \neg\textit{empty}(active queue) DO
06.     Pick and remove rule \( r_i \) from active queue.
07.     IF \textit{suspend}[i] THEN
08.         \textit{suspend}[i] := \text{FALSE};
09.         \textbf{restart} \textit{Index}_i;
10.      ELSE \textbf{start} \textit{Index}_i;
11.     ENDIF;
   {\textit{Index}_i} will return control here either when it terminates
   (i.e. with failure or success) or when it suspends. Handle
   each case appropriately.}
12.    IF \textit{fail}[i] THEN
   { In case of failure continue with next rule. }
13.       \textbf{goto} next iteration
14.    ELSIF \textit{suspend}[i] THEN

15. IF defined(G[pg[i]].fringenode) THEN
   place ri at the first uninspected child of G[pg[i]].fringenode;
16. ELSE
   { If G[pg[i]].fringenode is undefined then by skipping subtree rooted at this node Indexi completes the selection of ri. }
18. Select ri with success;
19. ENDIF
20. ELSE
   { Indexi has terminated successfully. }
   Select ri with success;
21. ENDIF
22. END
23. { Now the active queue is empty. So check the suspend queue.}
24. IF ~empty(suspend queue) THEN
   {All rules that have not failed are suspended. Therefore we need to skip portions of goal tree. Also the current goal string terminates as we are going to skip goal nodes.}
25. FOR i := endg DOWN TO 1 DO
26.   IF defined(G[i].varposn) THEN exit loop
27.   ELSE G[i].varposn := endg + 1;
28.   ENDIF
29. END;
30. { Instruct DemandScan to enter skip mode and restart it }
31. abort := TRUE;
32. resume DemandScan;
33. END;
END.

1. Correctness and Complexity

Let G be the goal array obtained by scanning it completely. Observe that some sequences of nodes in G may not appear in G constructed by the demand-driven algorithm. These nodes belong to the skipped subtrees.

The proof of correctness can be simplified by relating operations performed by Indexi with G to those performed by Index with G. Specifically, we must show that Indexi compares a node v in the goal with a node u in ri iff the same pair is compared by Index. We can do this by establishing a one-to-one correspondence between string-matching operations performed by Index and Indexi. For this purpose, we partition G into a sequence s1x1y1s2x2y2⋯sixiyi, where si contains functors used in a string-matching operation, gi is a substitution for a rule
variable, and \( x_i \) and \( y_i \) contain a sequence of goal variables that took substitutions. Note that some \( s_i, x_i, \) and \( y_i \) can be empty. We can similarly view \( \mathcal{G} \) as a sequence \( s'_1 x'_1 g'_1 y'_1 s'_2 x'_2 g'_2 \ldots s'_n x'_n g'_n y'_n. \) By establishing \( s_i = s'_i, \ x_i = x'_i, \) and \( y_i = y'_i \) we can show that replacing \( g_i \) by \( g'_i \) in \( G \) yields \( \mathcal{G} \). We accomplish this by proving the following results:

1. The sequence of nodes appearing in \( s_i \) have successive preorder numbers.
2. The node in \( G \) following the last node in \( s_i \) is its preorder successor.
3. The sequence of nodes that appear in \( x_i \) or \( y_i \) have successive preorder numbers.
4. The node in \( G \) following the last node in \( g_i \) is the preorder successor of the last node in \( g'_i \).

Using the above results it can be readily seen that replacing \( g_i \) by \( g'_i \) yields array \( G' \) in which adjacent nodes have successive preorder numbers and hence it must be \( \mathcal{G} \). We now prove results 1-4.

We say \( \text{node}(G[i]) = n \) iff the \( i \)th record in \( G[i] \) contains information about goal node \( n \). Recall the function \( \text{string} \) defined following the proof of Lemma A.2.

**Lemma B.1 (Result 1).** If \( \text{string}(G[i,j]) \) is used in a string match operation in procedure Index, then \( \text{node}(G[m+1]) \) is the preorder successor of \( \text{node}(G[m])(i < m < j) \).

**Proof.** Let \( u_l = \text{node}(G[l]) \) \((i \leq l \leq j)\). Assume that only \( u_i, u_{i+1}, \ldots, u_{i+n} \) \(((i + n) < j)\) have successive preorder numbers. Suppose \( w_1 \) is the preorder successor of \( u_{i+n} \) in the goal tree. There are two cases:

**Case 1.** \( u_{i+n} \) is a leaf (see Figure 17(a)). Since \( w_1, w_2, \ldots, w_q \) (and their descendants) do not appear in \( G \), the subtree rooted at these nodes must have been skipped by procedure DemandScan. For this to happen, abort must be true and no rule should be suspended at these nodes. Therefore, rule \( r_i \) must also be suspended at a node that appears after \( w_q \) in preorder. This means \( u_{i+n} \) is in the subtree computed as substitution for a variable

**FIGURE 17.** Situation used in the proof of Lemma B.1.
FIGURE 18. Situation used in the proof of Lemma B.3.

in \( r_i \) and hence cannot be a node involved in a string-matching operation for \( r_i \)--a contradiction. Therefore, \( q = 0 \) and \( u_{i+n+1} \) is the preorder successor of \( u_{i+n} \).

**Case 2.** \( u_{i+n} \) is not a leaf (see Figure 17(b)). Proof for this case is similar to case 1 above.

**Corollary B.1** (Result 2). If string \((G[i, j])\) is used in a string-matching step, then \( node(G[j+1]) \) is the preorder successor of \( node(G[j]) \).

**Proof.** Follows from the proof of above lemma. \( \square \)

**Lemma B.2** (Result 3). If procedure Indexi computes a substitution for goal variable at \( G[i] \), then \( node(G[i+1]) \) (if it exists) is the preorder successor of \( node(G[i]) \).

**Proof.** Note that Indexi cannot be suspended when computing substitutions for goal variables. Therefore, abort cannot be true. This implies no node will be skipped by DemandScan and hence the result. \( \square \)

**Lemma B.3** (Result 4). If subtree rooted at \( node(G[i]) \) is substituted for a variable in some rule \( r_j \), then \( node(G[G[i].subtree + 1]) \) is the preorder successor of the last node in the subtree rooted at \( node(G[i]) \).

**Proof.** Let \( u = node(G[i]) \) and \( v = node(G[G[i].subtree + 1]) \). Further, let \( x \) be the last node (in preorder) in the subtree rooted at \( u \) (see Figure 18). Suppose \( y \) is the preorder successor of \( x \) and \( y \neq v \). Let \( w \) be the parent of \( y \). Observe that the subtree field of \( G[i] \) is set in the last step in DemandScan\((u)\) just prior to return (see line 43). Now note that, at this point, there cannot be any uninspected nodes between \( u \) and \( w \) in goal \( g \) (otherwise \( y \) cannot be the preorder successor of \( x \)). Therefore, the return from DemandScan\((u)\) will result in a sequence of returns of DemandScan from these intermediate nodes. Clearly, these returns cannot add any new nodes to \( G \). This means when the recursive call DemandScan\((z)\) made in DemandScan\((w)\) returns, the last node in \( G \) will be \( node(G[G[i].subtree]) \). Now DemandScan\((w)\) will return without adding \( y \) to \( G \) only if abort is true and no
rule is suspended at \( y \) (see lines 31–37 in \textit{DemandScan}). Observe that \textit{abort} is set to true only by \textit{Select} and it does so only when all rules are suspended (see lines 24–32 in \textit{Select}). Furthermore, if no rule is suspended at \( y \), then all rules must be suspended at nodes that appear after \( y \) in preorder. Clearly, this means that the subtree rooted at \( u \) cannot be a substitution for any variable—a contradiction. Therefore, \( y = v \). □

We construct \( G' \) as follows. Replace each \( g_i \) by the preorder sequence of the subtree rooted at the first node in \( g_i \).

\begin{lemma}
\( G' = G \).
\end{lemma}

\begin{proof}
From results 1, 2, 3, and 4 it follows that adjacent nodes \( G' \) have successive preorder numbers and so \( G' = G \). □
\end{proof}

We have now shown that \textit{Index}_i compares a node \( v \) in the goal with a node \( u \) in \( r_i \) iff the same pair is compared by \textit{Index}. Therefore, from the correctness of procedure \textit{Index}, we have the following lemma.

\begin{lemma}
Procedure \textit{Index}_i terminates with \textit{fail} = \textit{false} iff \( r_i \) unifies (modulo nonlinearity) with the goal.
\end{lemma}

In what follows we establish the time complexity of the demand-driven algorithm. Here again we use \( l_g \) and \( k_g \) to denote the number of leaves and the number of variable occurrences in the goal, respectively. Similarly, let \( l_i \) and \( k_i \) denote the number of leaves and the number of variable occurrences in rule \( r_i \). Let \( m \) be the total number of rules.

\begin{lemma}
The demand-driven algorithm requires at most \( O(|G| + \sum_{i=1}^{m} \min\{k_g + k_i, l_g, l_i\}) \) time.
\end{lemma}

\begin{proof}
Observe that procedure \textit{Index}_i contains two loops—lines 15–19 and lines 21–90. The steps that are not part of these two loops take only constant time. So complexity of \textit{Index}_i is given by the sum of the time taken by these two loops. Each iteration of the first loop resumes a suspended \textit{DemandScan}. In our analysis we account for the cost of each iteration of this loop as part of the cost of \textit{DemandScan} restarted by that iteration. We also make a similar accounting of costs of inner loops at lines 59–63 and 75–79 that are part of the second loop in \textit{Index}_i. Following this amortization, the complexity of \textit{Index}_i now depends only on the complexity of the second loop without lines 59–63 and 75–79. As procedure \textit{Index} (see Section 2.5), \textit{Index}_i also executes the inner loop (lines 21–35) at least once for each iteration of the outer loop. Since we have eliminated the cost of lines 56–70 and 71–75, it follows that the complexity of \textit{Index}_i depends only on the complexity of the inner loop (because other steps now take only constant time). Since the steps in this inner loop take only constant time and each iteration computes one substitution, we conclude that the complexity of \textit{Index}_i is bounded by the number of substitutions computed in it which is at most \( O(\min\{k_i + k_g, l_i, l_g\}) \) (by Theorem A.2). Selecting \( m \) rules requires \( m \) invocations of \textit{Index}_i. All these invocations together require \( O(\sum_{i=1}^{m} \min\{k_j + k_g, l_j, l_g\}) \).
Next we analyze the complexity of procedure \textit{Select}. Clearly, the complexity of \textit{Select} depends on the complexity of the loop in lines 4–33, which in turn depends on the complexity of the two inner loops. Because each iteration of the inner while loop restarts an invocation of \textit{Index}_i, the total number of iterations of this loop over all iterations of the outer loop is bounded by the number of times procedure \textit{Index}_i is restarted. Observe that procedure \textit{Index}_i suspends only at line 32 and upon restarting it computes a substitution. Therefore, the total number of iterations of inner while loop is at most \(O\left(\sum_{j=1}^{m} \min\{k_j + k_g, l_j, l_g\}\right)\). Since lines 6, 16, and 18 can be implemented in \(O(1)\) time, the complexity of the inner while loop is \(O\left(\sum_{j=1}^{m} \min\{k_j + k_g, l_j, l_g\}\right)\). Each successful iteration of the for loop in lines 25–29 updates a record in \(G\) with uninitialized \textit{varposn} field. Therefore, the total number of successful iterations of this for loop is bounded by \(|G|\). Observe that after each unsuccessful iteration of this loop, \textit{Select} restarts \textit{DemandScan} (line 31) which will add a new record to \(G\). Therefore, the total number of unsuccessful iterations is again bounded by \(O(|G|)\). Since lines 6, 16, and 18 can be implemented in \(O(1)\) time, the complexity of the inner while loop is \(O\left(\sum_{j=1}^{m} \min\{k_j + k_g, l_j, l_g\}\right)\).

Each successful iteration of the for loop in lines 25–29 updates a record in \(G\) with uninitialized \textit{varposn} field. Therefore, the total number of successful iterations of this for loop is bounded by \(|G|\). Observe that after each unsuccessful iteration of this loop, \textit{Select} restarts \textit{DemandScan} (line 31) which will add a new record to \(G\). Therefore, the total number of unsuccessful iterations is again bounded by \(O(|G|)\). Since lines 6, 16, and 18 can be implemented in \(O(1)\) time, the complexity of the inner while loop is \(O\left(\sum_{j=1}^{m} \min\{k_j + k_g, l_j, l_g\}\right)\).

The steps inside the for loop take only constant time. So the complexity of the for loop over all iterations of the outer while loop is \(O(|G|)\). Therefore, the worst-case complexity of \textit{Select} is \(O\left(|G| + \sum_{j=1}^{m} \min\{k_j + k_g, l_j, l_g\}\right)\).

Finally, we analyze the complexity of \textit{DemandScan}. Since lines 1 and 2 add new symbols to \(G\), it follows that the total number invocation of \textit{DemandScan} is bounded by \(|G|\). Note that each invocation of \textit{DemandScan} is suspended and resumed exactly once (see lines 10, 11, and 21). Therefore, the total number of suspensions and restarts of \textit{DemandScan} is also bounded by \(O(|G|)\). Using transition tables to represent the automaton, we can implement \textit{ACScan} so that each of its invocations takes constant time. Therefore, all lines except those in the for loops in lines 5–9 and 25–38 take only constant time. In view of the discussion in the preceding paragraph, the complexity of the for loop in lines 5–9 over all invocations of \textit{DemandScan} is bounded by \(O(|G|)\). (In fact the sum of the complexities of the for loops at line 5 in \textit{DemandScan} and line 25 in \textit{Select} together is \(O(|G|)\).) Observe that each successful iteration of the loop in lines 25–38 takes constant time and invokes \textit{DemandScan} once recursively. Therefore, the complexity of successful iterations of this loop over all invocations of \textit{DemandScan} is again bounded by \(O(|G|)\). Since the loop fails at most once in each \textit{DemandScan}, it follows that the computations performed in all invocations of \textit{DemandScan} are bounded by \(O(|G|)\). Recall that during analysis of \textit{Index}_i, by amortization, we added to the complexity of \textit{DemandScan} an amount of time that is proportional to the number of times each \textit{DemandScan} is restarted. Because each \textit{DemandScan} is restarted only once, this overhead does not affect the asymptotic complexity of \textit{DemandScan}.

Thus demand-driven indexing of \(m\) rules \(r_1, r_2, \ldots, r_m\) takes at most \(O(|G| + \sum_{j=1}^{m} \min\{k_j + k_g, l_j, l_g\})\) time. \(\square\)

**APPENDIX C. PROOF OF CORRECTNESS OF THE TrieSel ALGORITHM**

In the rest of this section we formulate necessary concepts and prove the correctness of the \textit{TrieSel} algorithm. In particular, we will show that \textit{TrieSel} terminates with a set containing only the rules that unify (modulo nonlinearity) with the goal.

In the following discussions we assume \(t_i\) denotes the trie constructed from rule \(r_i\). We also assume that edges in term trees are labeled by integers such that the edge leaving a node toward its \(i\)th child is labeled by \(i\). Let \(v_0, v_1, v_2, \ldots, v_l = v\) be
the sequence of nodes that appear in the root-to-node path of \( v \) in rule \( r \). Further, let \( \text{label}(x) \) and \( \text{label}(x, y) \) denote the label of node \( x \) and the label of the edge between nodes \( x \) and \( y \).

**Definition C.1.** The path to a node \( v \), denoted by \( \text{path}(v) \), in a tree or a trie is the string obtained by concatenating the edge labels in the path from root to \( v \).

**Definition C.2.** The labeled path up to \( v \), denoted by \( \text{lp}(v) \), is the sequence \( \text{label}(v_0) \circ \text{label}(v_0, v_1) \circ \cdots \circ \text{label}(v_{l-1}, v_l) \). If \( v \) is labeled by a functor \( f \) and \( \text{lp}(v) = \alpha \), then labeled path including \( v \), denoted by \( \text{lp}(\overline{v}) \) is \( \alpha \circ f \). On the other hand, if \( v \) is labeled by a variable, then \( \text{lp}(v) = \text{lp}(\overline{v}) \).

For example, in Figure 1, \( \text{lp}(11) \) and \( \text{lp}(ll) \) are \( f2g1 \) and \( f2g1\alpha \), respectively.

Observe that given rule \( r \) and its trie \( t \), the construction ensures that labeled paths (i.e., \( \text{lp}(v) \) and \( \overline{\text{lp}(v)} \)) in \( r \) correspond to paths in \( t \) and vice versa. More importantly, if node \( v \) (labeled by a functor) in \( r \) is split into node pair \( \langle x_1, x_2 \rangle \) in the construction of \( t \), then \( \text{lp}(v) = \text{path}(x_1) \) and \( \overline{\text{lp}(v)} = \text{path}(x_2) \). The following definition constructs a mapping between nodes in \( t \) to those in \( r \). Let \( x \) and \( v \) be nodes in \( t \) and \( r \), respectively. Recall the definition of \( \text{pos} \) given in Definition A.2.

**Definition C.3.** \( \text{node}_t(x) = v \) iff \( \text{pos}(v) \) in \( r \) is equal\(^6\) to the string obtained by dropping functor labels in \( \text{path}(x) \).

For the trie \( t_1 \) in Figure 9, both \( \text{node}_t(5) \) and \( \text{node}_t(6) \) denote the node 11 in rule \( r_1 \) (see Figure 1). Observe that for node \( v \) in \( r \) and nodes \( x \) and \( y \) in \( t \), if \( \text{lp}(v) = \text{path}(x) \) and \( \overline{\text{lp}(v)} = \text{path}(y) \), then \( \text{node}_t(x) = (\text{node}_t(y)) = v \). Recall that in the construction of \( t \) a node \( v \) in \( r \), labeled by a functor, is split into a pair of nodes and this definition relates \( v \) to them.

**Definition C.4.** A node in \( t \) is said to be active if its outgoing edge is labeled by a functor.

Observe that if \( x \) is active, then \( \text{lp}(\text{node}_t(x)) = \text{path}(x) \). On the other hand, if \( y \) is the child of an active node, then \( \overline{\text{lp}(\text{node}_t(y))} = \text{path}(y) \).

The trie \( T \) is similar to the goto trie of the Aho–Corasick automaton and has the following properties.

\( P_1: \) Every prefix of a path in a rule trie is represented by a unique node in \( T \). Specifically, \( \alpha \) is a path in \( T \) iff it is a path in a rule’s trie. Moreover, if \( \beta \) is a root-to-leaf path in \( T \), then it cannot be a path of a nonleaf node in any rule’s trie.

\( P_2: \) Outgoing edges from any node are either all labeled with functors or all labeled with integers. In the former case, the node is active.

\( P_3: \) \( S_v \cap M_v = \emptyset \). Furthermore if \( v \) is a leaf, then \( S_v = \emptyset \).

Observe that a node in \( T \) is obtained by coalescing together (some) nodes of rule tries. Consequently, a single node in \( T \) may correspond to nodes in several rules.

\(^6\)For comparing paths with \( \text{pos} \), the period (.) in \( \text{pos} \) is ignored.
(at most one node per rule). Let \( \text{note}_T(v, r) \) denote the node in \( r \) that corresponds to \( v \) in \( T \). Formally,

**Definition C.5.** If \( r \in S_v \cup M_v \) and \( x \) is a node in \( t \), then \( \text{note}_T(v, r) = \text{node}_t(x) \) iff \( \text{path}(x) = \text{path}(v) \).

For example, \( \text{note}_T(3, r_2) \) for node 3 in Figure 10 is node 13 in Figure 1. In our proof we need the following properties of \( S_v \) and \( M_v \).

**Proposition C.1.** Let \( w_1, w_2, \ldots, w_l \) be the children of an active node \( v \). Then \( r \in S_{w_i} \cup M_{w_i} \) iff \( r \in S_v \) and \( \text{note}_T(v, r) \) has the same label as the outgoing edge from \( v \) to \( w_i \).

**Proof.** Let \( t \) be the trie constructed from \( r \) and let \( f \) be the label on the edge from \( v \) to \( w_i \). By construction of \( T \), \( r \in S_{w_i} \cup M_{w_i} \) iff \( \text{path}(v) \circ f = \text{path}(x) \) for some node \( x \) in \( t \). This can happen iff there are two nodes \( x \) and \( y \) (both in \( t \)) such that \( x \) is a child of \( y \) with \( \text{path}(x) = \text{path}(v) \circ f \) and \( \text{path}(y) = \text{path}(v) \). Once again, by construction of \( t \), this can happen iff there is a node \( q \) in \( r \) such that \( \text{lp}(q) = \text{path}(y) \) and \( \text{lp}(q) = \text{path}(x) \). Now by Definition 2, \( \text{label}(q) = f \). Again by Definition 5, \( \text{note}_T(v, r) = q \) and hence the lemma. \( \square \)

**Proposition C.2.** If \( w_1, w_2, \ldots, w_l \) are the children of node \( v \) in \( T \) that is not active, then \( S_v = M_{w_i} \cup S_{w_i} \).

**Proof.** Let \( r \in M_{w_i} \cup S_{w_i} \). By construction, \( \exists x \) in \( t \) such that \( \text{path}(x) = \text{path}(w_i) \). So all prefixes of \( \text{path}(w_i) \) are also root-to-node paths in \( t \). Moreover, proper prefixes of \( \text{path}(w_i) \) must be paths to nonleaf nodes. Therefore, \( r \in S_v \), i.e., \( M_{w_i} \cup S_{w_i} \subseteq S_v \).

Let \( z \) be the parent of \( v \) and let \( f \) be the label of the outgoing edge from \( z \) to \( v \). Now suppose \( r \in S_v \). By Proposition C.1, it follows that \( \text{note}_T(v, r) \) for any \( r \in S_v \) is labeled by \( f \). Therefore, all such nodes must have \( l \) children (by uniqueness of functor arity). Hence, \( \text{path}(v) \circ 1, \text{path}(v) \circ 2, \ldots, \text{path}(v) \circ l \) are paths in trie \( t \). Hence, \( r \in M_{w_i} \cup S_{w_i} \), i.e., \( S_v \subseteq M_{w_i} \cup S_{w_i} \), and so \( S_v = M_{w_i} \cup S_{w_i} \).

**Proposition C.3.** If \( v \) is a nonleaf node in \( T \) that is not active, then \( M_v = \emptyset \).

**Proof.** Suppose \( r \in M_v \). Let \( z \) be the parent of \( v \) with \( f \) as the label on the edge from \( z \) to \( v \). Since \( v \) is nonleaf, the arity of \( f \) is nonzero. Therefore, any node labeled with \( f \) cannot be a leaf node. In particular, \( \text{note}_T(v, r) \) is not a leaf node in \( r \). Therefore, \( r \not\in M_v \), a contradiction. \( \square \)

**Lemma C.1.** Suppose \( r \in S \) in the call TrieSel\((u, v, S)\). Then \( S \subseteq S_v \cup M_v \).

**Proof.** By induction on depth of recursion.

**Base case:** The initial call to TrieSel is started at root \( z \) of \( T \) with \( S \) initialized to the set of all rules. Since \( z \) is a root of \( T \), \( \text{path}(z) \) is an empty string. Hence, \( S_z \cup M_z \) is the set of all rules. Therefore, \( S \subseteq S_z \cup M_z \).

**Induction step:** Assume that the lemma holds up to recursive calls of depth \( k \).

Let TrieSel\((u_1, v_1, X_1)\) be a recursive call of depth \( k + 1 \). Furthermore, let
\( \text{TrieSel}(u, v, X) \) be the call that invoked \( \text{TrieSel}(u_1, v_1, X_1) \). From lines 3 and 6 in Figure 11, it is clear that \( v \) is the parent of \( w \), and \( w \) in turn is the parent of \( v_1 \). From lines 5 and 8 in the algorithm we conclude that \( X_1 \subseteq X \cap (S_w \cup M_w) \subseteq S_w \cup M_w \). Now observe that \( w \) is a nonleaf node that is not active. Therefore, by Proposition C.3, \( M_w = \emptyset \). Further by Proposition C.2, \( S_w = S_{v_1} \cup M_{v_1} \). Hence \( X_1 \subseteq S_w \cup M_w = S_w = S_{v_1} \cup M_{v_1} \).

We now prove the correctness of \( \text{TrieSel} \). Here we use the fact that all calls to \( \text{TrieSel} \) are initiated only from active nodes.

**Lemma C.2.** Suppose \( S_{\text{out}} \) is the set of rules returned from \( \text{TrieSel}(u, v, S_{\text{in}}) \). \( r \in S_{\text{out}} \) iff \( r \in S_{\text{in}} \) and the subtree rooted at \( \text{note}_T(v, r) \) unifies (modulo nonlinearity) with the subtree rooted at \( u \) in the goal.

**Proof.** By induction on the height of \( v \). Let \( q_r \) denote the subtree of rule \( r \) rooted at \( \text{note}_T(v, r) \). Similarly, let \( q_u \) denote the subtree of the goal rooted at \( u \).

**Base case:** Height of \( v \) is 1. (Note that \( v \) is an active node; therefore, its height cannot be 0.) Now we have three cases to consider depending upon the label of \( u \) and labels of outgoing edges from \( v \).

**Case 1.** \( u \) is labeled by a variable. Clearly, for any rule \( r \in S_{\text{in}} \), the subtree rooted at \( \text{note}_T(v, r) \) will unify (modulo nonlinearity) with that rooted at \( u \). Observe that by line 1, \( \text{TrieSel} \) returns \( S_{\text{in}} \) unchanged when \( u \) is labeled by a variable. Therefore, the lemma holds in this case.

**Case 2.** \( u \) is labeled by a functor, say \( f \), that does not appear on any outgoing edge from \( v \). Because \( S_{\text{in}} \subseteq S_v \cup M_v \) (by Lemma C.1) and no edge leaving \( v \) is labeled by \( f \), from Proposition C.1, if \( r \in S_{\text{in}} \), then \( \text{note}_T(v, r) \) is labeled by either a variable or by a functor other than \( f \). Therefore, subtree rooted at \( \text{note}_T(v, r) \) can unify with that rooted at \( u \) iff \( \text{note}_T(v, r) \) is labeled by a variable, i.e., \( \text{note}_T(v, r) \) must be a leaf node and, hence, \( r \in M_v \). Observe, by line 11 in \( \text{TrieSel} \), that

\[
S_{\text{out}} = S_1 \cup M_1
= (S_{\text{in}} \cap S_v) \cup (S_{\text{in}} \cap M_v)
\]  
(by lines 2, 3, 10 and assumption on \( u \) and \( v \) for this case.)

\[
= S_{\text{in}} \cap M_v
\]  
(since \( S_v \cap M_v = \emptyset \) and \( S_{\text{in}} \subseteq M_v \cup S_v \).)

**Case 3.** \( f \) is the label of \( u \) and also of the outgoing edge from \( v \) to \( w \). Because the height of \( v \) is 1, \( w \) must be a leaf in \( T \). So by property P1, \( \text{note}_T(w, r) \) is a leaf for any \( r \in M_w \cup S_w \). In other words, \( f \) is a functor of arity 0. This means \( u \) is also a leaf in the goal. Clearly, if \( q_u \) unifies (modulo nonlinearity) with \( q_r \), then \( q_r \) is a single node tree labeled either with a variable or the functor \( f \). Observe that if \( r \in S_{\text{in}} \) and the (only) node in \( q_r \) is labeled by a variable, then \( r \in M_v \) (because \( S_{\text{in}} \subseteq S_v \cup M_v \)). On the other hand, if the node in \( q_r \) is labeled by \( f \), then \( r \in M_w \cup S_w \). Therefore, the lemma in this case is proved if \( S_{\text{out}} = (S_{\text{in}} \cap (M_w \cup S_w)) \cup (S_{\text{in}} \cap M_v) \).


lines 1, 5, 6, and 11, TrieSel in fact returns \((S_{in} \cap (M_{w} \cup S_{w})) \cup (S_{in} \cap M_{v})\) in this case.

**Induction step:** We assume that the lemma holds for recursive calls made at nodes of \(T\) such that the height of the subtrie (of \(T\)) rooted at that node is less than \(k\). Let \(k\) be the height of the subtrie rooted at \(v\). Again we divide the proof into three cases as above.

**Case 1.** Similar to the base case.

**Case 2.** Similar to the base case.

**Case 3.** \(f\) is the label of \(u\) and also of the outgoing edge from \(v\) to \(w\). Furthermore, let \(w\) have \(m\) children. So the arity of \(f\) is \(m\) and, therefore, \(u\) also has \(m\) children. Now define sets of rules \(R, R_0, R_1, \ldots, R_m\) as follows.

\[
R = \{r \in S_{in} \mid \text{note}_{T}(v, r) \text{ is labeled by a variable}\},
\]

\[
R_0 = \{r \in S_{in} \mid \text{note}_{T}(v, r) \text{ is labeled by } f\},
\]

\[
R_i = \{r \in R_{i-1} \mid \text{ith subtree of } q_r \text{ unifies (modulo nonlinearity) with the ith subtree in } q_u\}.
\]

Observe that the lemma in this case is established if \(S_{out} = R \cup R_m\). Clearly, \(R = M_1\) (by line 2). Let \(X_0, X_1, \ldots, X_m\) be the values of \(S_1\) at the end of the \(i\)th iteration of the for loop in lines 6–9. We show by induction on \(i\) that \(X_i = R_i\). Note \(S_1 = X_0\) prior to entry of the for loop. By line 5, \(X_0 = S_{in} \cap (M_{w} \cup S_{w})\). By Proposition C.1, \(R_0 = X_0\). Now assume that at the end of \(l\)th iteration, \(X_l = R_l\). In the \((l + 1)\)th iteration we invoke TrieSel recursively iff \(w_{l+1}\) is a nonleaf node. Suppose we invoke TrieSel. Then by induction hypothesis, \(r \in X_{l+1}\) iff \(r \in X_l\) and the \((l + 1)\)th subtrees of \(q_r\) and \(q_u\) unify (modulo nonlinearity). Therefore, \(X_{l+1} = R_{l+1}\). On the other hand, if we did not make a recursive call, then there is no edge leaving \(w_{l+1}\), indicating that the \((l + 1)\)th subtree of all rules in \(S_{w}\) (and hence \(X_l\)) is a single node subtree labeled by a variable. Therefore, all these subtrees unify with the \((l + 1)\)th subtree in \(q_u\). Hence, \(R_l = R_{l+1}\). Therefore, \(X_l = X_{l+1} = R_l = R_{l+1}\). In conclusion, at the end of the \(m\)th iteration, \(S_1 = R_m\) and hence \(S_{out} = R \cup R_m\) by line 11.

**Theorem C.1.** TrieSel selects a rule iff it unifies (modulo nonlinearity) with the goal.

**Proof.** Since the initial call to TrieSel is made with the set containing all rules with \(u\) and \(v\) as the roots of goal and \(T\), this theorem is a direct consequence of the Lemma C.2. □

Recall that merging rule tries into \(T\) is carried out iteratively. In the \(i\)th iteration, we merge \(T_i\) with \(T_{i-1}\). Had we implemented this using a naive approach that inserts each root-to-leaf in the \(T_i\) path independently, then this insertion would take \(O(|t_i|^2)\) time. However, [8] shows how to insert in \(O(|t_i|)\) time. Since \(|t_i|\) is \(O(|r_i|)\), using [8] we can construct \(T\) in time linear in the sum of the sizes of the rules. While scanning the goal in each recursive call, we update \(S_1\) and \(M_1\). By associating a bit per rule, these sets can be represented as bit vectors, and the intersection
and union can now be done in $O(1)$ time by ANDing and ORing. This means the running time of the selection is proportional to the number of recursive calls, which is bounded by the size of the portion of the goal scanned.

REFERENCES
