Theoretical Computer Science 254 (2001) 187–223

Theoretical Computer Science

www.elsevier.com/locate/tcs

Automata-driven efficient subterm unification ☆

R. Ramesh[a, 1], I.V. Ramakrishnan[b,*,2], R.C. Sekar[b]

[a] *Box 660199, MS 8645, Texas Instruments Inc., Dallas, TX 75266, USA*

[b] *Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794, USA*

Received December 1996; revised January 1999

Communicated by O.H. Ibarra

Abstract

Syntactic unification has widespread use in computing. There are several operations used in deductive computing such as critical pair generation, paramodulation and narrowing that require unifying a term s with every subterm of another term p. This subterm unification problem can be solved naively by repeatedly unifying s with each subterm of p in isolation. The drawback of doing unification in isolation is that commonality among subterms of p is ignored. We present an algorithm for efficient subterm unification by exploiting this commonality. The central idea used in our algorithm is to reduce the common part computation in unification into a string-matching problem and solve it efficiently using a string-matching automaton. The automaton succinctly captures the commonality between subterms of p. The string-matching approach, in conjunction with two new techniques called *bidirectional-reduce* and *marking* enables efficient unification of s with every subterm of p.

Keywords: Syntactic unification; Subterm unification algorithm; String-matching automata

1. Introduction

Syntactic unification is a ubiquitous operation in computing. Many deductive computing applications require unifying a term s with subterms of another term p. For instance, in Knuth–Bendix completion procedure, given a set of rewrite rules $l_1 \to r_1, l_2 \to r_2, \ldots, l_n \to r_n$, critical pairs are generated by unifying each l_i $(1 \leqslant i \leqslant n)$ with subterms of ev-

☆ A preliminary version of this paper appeared in the Proceedings of the 1994 FST & TCS.

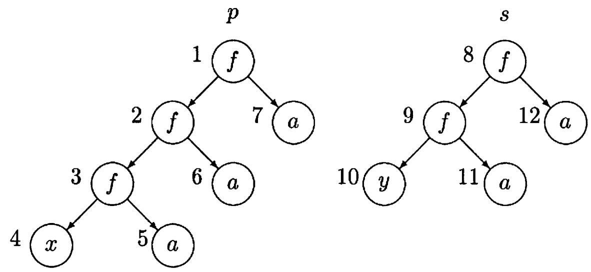
* Corresponding author.

E-mail addresses: rramesh@dadd.ti.com (R. Ramesh), ram@cs.sunysb.edu (I.V. Ramakrishnan), sekar@cs.sunysb.edu (R.C. Sekar).

[1] Research completed while at the Department of Computer Science, University of Texas at Dallas and supported by NSF grant CCR-9110055.

[2] Research supported by NSF grants CCR 9102159, 9404921, 9510072, 9705998, 9711386 and INT 9600598.

0304-3975/01/$ - see front matter
PII: S0304-3975(99)00131-0



Node label comparisons in unification of s with

Subterm rooted at 1	Subterm rooted at 2
1 with 8	2 with 8
2 with 9	3 with 9
6 with 11	5 with 11
7 with 12	6 with 12

Fig. 1. Unification of p with subterms of s.

ery l_j ($1\leqslant i\leqslant n$). The technique of paramodulation used in resolution-based theorem provers for handling equality and the narrowing operation used in combining logic and functional programming, also require unifying s with subterms of p.

This problem, henceforth referred to as *subterm unification*, can be done naively by repeatedly unifying s with every subterm of p. Each of these unifications is independent of any previous operations, i.e., they are done in isolation. However note that: (1) s is a common term in all these unifications and (2) many (or all) of these unifications involve overlapping subterms of p. The drawback of doing unification in isolation is that we do not exploit these commonolities and hence repeat computations that are common. We illustrate through Fig. 1 the opportunities for optimization. To unify s with p at its root we must compare labels of nodes 1,2,6 and 7 with that of 8,9,11 and 12, respectively. To unify s with subterm of p rooted at node 2 we have to compare the labels of nodes 2,3,5 and 6 with that of 8,9,11 and 12 respectively. Note that the nodes 2 and 6 are common to the two unifications. Doing unifications in isolation will result in inspecting these two nodes twice. Observe that from the first unification we know that nodes 1, 2 and 8 have the same labels. Therefore, we need not compare labels of nodes 2 and 8 in the second unification. Similarly we can avoid comparing labels of nodes 6 and 12. Moreover, based on the examination of node labels at 5,6, and 7 when attempting unification at nodes 1 and 2, we can conclude that unification at nodes 5, 6 and 7 is bound to fail. Early detection of such non-unifiable subterms can lead to further savings in time.

There has been considerable research in factoring out common computations for pattern matching[3] (e.g. [3, 4, 9]). In [10] we gave an algorithm for factoring out com-

[3] In pattern matching p is always ground and s is typically linear.

mon computations that arise in indexing of Prolog clauses. [4] However, the design of a similar efficient algorithm for the subterm unification problem has remained open and forms the topic of this paper.

1.1. Summary of results

The main contribution of this paper is an efficient subterm unification algorithm to unify s with subterms of p that exploits commonality among subterms. Our algorithm, following Martelli and Montanari's approach [6], does unification by solving term equations. The basic operations in this algorithm are computing the common part of terms and substitutions for variables, as described in Section 2. We transform the common part computation into a highly structured string-matching problem. This structure enables us to construct an automaton (by preprocessing s and p) to efficiently solve the string-matching problem. The automaton succinctly represents commonality between subterms of p and enables efficient unification of s with each of these subterms without examining any symbol in s or p more than once. Using it we compute common part in time proportional to number of variables in the terms (see Section 3). To efficiently compute substitutions for variables, we introduce the *bidirectional-reduce* operation which reduces the number of intermediate substitutions (see Sections 3 and 4). If sharing among terms in substitutions is not handled then it can result in excessive reexamination of the same variable occurrence. We use a *marking* technique, in Section 5, that introduces virtual variables to facilitate sharing without affecting the string-matching automaton. We integrate these techniques to efficiently perform subterm unification in Section 6. Our subterm unification algorithm generalizes (i) our tree pattern matching algorithm in [9] by allowing variables in p and (ii) our Prolog indexing algorithm in [10] by allowing both s and p to be nonlinear and doing unifications at all the nonroot positions of p.

Our algorithm first preprocesses s and p into a string-matching automaton prior to subterm unification. The following is a summary of our complexity results.

- Constructing the string-matching automaton requires only $O(|s| + |p|)$ time. Note that in applications using subterm unification, p and s are created and destroyed at run time. Therefore it is crucial to have small preprocessing costs.
- Let t be any subterm in p and k denote the number of occurrences of variables in s and t. Let k_d denote the number of distinct variables in s and t. Table 1 is a summary of the worst-case running time for unifying any subterm t with s. (In the table, α denotes the inverse of Ackermann's function and *multiplicity* is the maximum over the number of occurrences of any variable.) In contrast the linear time algorithms in [7, 8] will always require $O(|s| + |t|)$ time to unify s with t. We also show that the asymptotic running time (including the cost of preprocessing) of our subterm unification is always better than doing independent unifications (see Section 6).

[4] The indexing algorithm assumes that both p and s are linear and matches are performed only at the root.

Table 1
Table of asymtotic complexities

t \ s	Linear	Nonlinear
Linear	O(k)	O(k)
Nonlinear	O(k)	$\frac{\mathit{multiplicity} \leqslant 2}{\mathrm{O}(k)}$ $\frac{\mathit{multiplicity} > 2 \text{ and } k_d\alpha(k_d) < \lvert s\rvert + \lvert t\rvert}{\mathrm{O}(\mathit{min}\{k_d k, \lvert s\rvert + \lvert t\rvert\})}$ $\frac{\mathit{multiplicity} > 2 \text{ and } k_d\alpha(k_d) \geqslant \lvert s\rvert + \lvert t\rvert}{\mathrm{O}(\lvert s\rvert + \lvert t\rvert)}$

Observe from Table 1 that the worst-case performance of our algorithm occurs only in the case when both terms are nonlinear and each one has a variable with more than two occurrences. But note that for any subterm t of p, the number of occurrences of any variable in it decreases as t's distance from the root of p increases. This implies that even in the worst-case scenario the performance of our algorithm will only improve as we unify s with subterms of p that are farther and farther away from the root.

2. Preliminaries

A *term* is either a variable or an expression of the form $f(t_1, t_2, \ldots, t_n)$ where f is a function symbol of arity $n \geqslant 0$ and $t_1, t_2, \ldots, t_n$ in turn are also terms. The notion of a *position* in a term is used to refer to subterms in a term as follows. A position is either the empty string Λ that reaches the root of the term or $\lambda.i$ (λ is a position in the term and i is an integer) which reaches the ith argument of the root of the subterm reached by λ. We use t/λ to refer to the subterm of t reached by λ. The development of our algorithm is based on Martelli and Montanari's algorithm in [6]. We sketch its high-level description below.

Their algorithm views unification as the problem of solving term equations of the form $s = t$ where s and t are the terms to be unified. It operates by repeatedly transforming the initial equation $s = t$ into an equivalent set of equations until either it detects that there is no unifier or when the resulting equations are in *solved form*. An equation $x = q$ (x is a variable) is an *elementary* equation and q is referred to as a *substitution* for x. A set of elementary equations $\{x_1 = t_1, x_2 = t_2, \ldots, x_n = t_n\}$ is said to be in solved form iff x_i does not appear in terms $t_i, t_{i+1}, \ldots, t_n$. Such a set is also said to be in *canonical* form. An equation queue E_x is associated with every variable x. E_x is the list of right-hand sides (rhs) of elementary equations whose left-hand sides (lhs) are x. The solution queue S has elementary equations in canonical form.

Each transformation step augments S by adding one more equation to it. This is done by identifying a variable x that does not occur in any term in any of the equation queues (this is the *occur-check*) and selecting its equation queue E_x for processing. Note that each term r in E_x represents the equation $x=r$. E_x is processed by decomposing the terms in it into a *common part* c and *frontier* F. Conceptually, we can view the common part of a collection of terms as follows. First superpose the terms at their roots. Next mark all those nodes that fall on variables. If the terms obtained by deleting the subtrees rooted at all the marked nodes are identical then they constitute the common part. Otherwise common part does not exist and unification fails. The deleted subterms constitute the frontier; e.g., if E_x contains the terms $f(x_1)$, $f(g(x_2))$ and $f(g(h(x_3)))$ then the common part will be $f(x_1)$ and the frontier will contain the equations $x_1=g(x_2)$ and $x_1=g(h(x_3))$. After computing the common part c and frontier F, S is augmented with $x=c$ and equations in F are processed as follows. For each elementary equations of the form $y=z$ in the frontier we merge equation queues E_y and E_z. The rhs of remaining equations are distributed to the equation queues of the variables appearing in the lhs.

Recall that prior to each transformation, we must select an equation queue for processing and that we select E_x if x does not occur in any term in any of the unprocessed equation queues. To efficiently perform this selection we keep a count of the number of occurrences of variables among the terms in the unprocessed equation queues. In the beginning the counters of all variables are initialized by counting their occurrences in s and t. After each successful transformation step the counters are updated prior to selecting the next equation queue. An equation queue E_x is selected when the occurrence counter of x becomes zero.

The unification process can be described using four procedures – *CommonPart* (to compute the common part and frontier), *MergeQ* (to merge equation queues), *AddToQ* (that distributes the rhs terms) and *UpdateCounters* (to maintain the counters). Procedure *Unif* in Fig. 2 is an outline of how these procedures are integrated to perform unification of s and t. Observe from line 16 that two equation queues E_y and E_z are merged if the equation $y=z$ is in the frontier. This operation is performed by procedure *MergeQ* to ensure that in subsequent steps x and y will be treated as the same variable. In general, we may compute several such equations and hence will merge several pairs of equation queues. To implement *MergeQ* operation, we choose a leader among the variables whose equation queues are to be merged. The new equation queue, obtained by merging equation queues of variables in the group, is the equation queue of the leader. Specifically, the new substitutions computed for variables in this group will be added to the leader's equation queue. Similarly, counters of all variables in this group will be added together and becomes the counter of the leader. This means that $AddToQ(q, E_y)$ must add the term q to the equation queue of y's leader. Similarly if the counter of y is to be updated then *UpdateCounters* will update the counter of y's leader.

Let T_{cp}, T_{update} and T_{merge} denote the time taken by procedures *CommonPart*, *MergeQ* and *UpdateCounters* over all iterations of the while loop between lines 6–27 (in Fig. 2). If T_{unif} is the time taken to unify s and t then:

Theorem 1. *T_{unif} is* $\mathrm{O}(T_{cp} + T_{update} + T_{merge})$

```
Procedure Unif(s, t)
begin
{EqnTodo contains the unprocessed equation queues.}
{firstQ is a dummy equation queue containing s and t.}
{S is the solution queue and initially it is empty.}
EqnTodo := {firstQ}
Initialize counters of the variables to the number of their occurences in s and t
while EqnTodo is not empty do
  if there is a variable whose counter is 0 then
    Select a variable whose counter is 0. Let it be x.
    Let E_x = {t_1, t_2, ..., t_1}
    if CommonPart(t_1, t_2, ..., t_1) exists then
      Let c and F be the commonpart and frontier respectively
      add x = c to S.
      for each equation e in F do
        Let y be the variable on the lhs of e
        if rhs of e is a variable, say z then
          MergeQ(y, z)
        else
          Let q be the term on the rhs of e
          AddToQ(q, E_y)
        endif
      end
    else return (failure)
    endif
    UpdateCounters()
    else return (failure)
    endif
  endwhile
end.
```

Fig. 2. Unification algorithm.

Proof. It is quite straightforward to implement *AddToQ* so that it takes time proportional to the number of substitutions computed. Therefore the running time of *AddToQ* over all invocations is never more than T_{cp}. Hence the result.

3. Computing common part efficiently

Observe that common part computation involves only comparing node labels and distributing the substitutions of variables into appropriate equation queues. The latter

```
begin
    fail := false
    Frontier := ∅
    repeat
       if S(p_s) and T(p_t) are both function symbols then
          {match phase}
          if S(p_s) ≠ T(p_t) then
             fail := true
          else
             p_s := p_s + 1
             p_t := p_t + 1
          endif
       elseif one of them is a variable, say S(p_s)
          {skip phase}
          Let x = S(p_s) and q be the subterm rooted at T(p_t)
          Frontier := Frontier ∪ {x = q}
          p_s := p_s + 1
          advance p_r to node immediately following the subtree rooted at T(p_t)
       end if
    until (fail=false) or (S and T are completely scanned)
end.
```

Fig. 3. Simple algorithm for computing common part.

operation (i.e. $AddToQ$) is quite simple to implement and only takes time proportional to number of substitutions computed. In [6], pairs of node labels are compared position by position and hence common part computation takes time proportional to the sum of the sizes of the terms. The key idea in our approach is to compare node labels in a sequence of positions in O(1) time. This enables us to compute the common part also in time proportional to the number of substitutions. We do this by reducing the common part computation into a string-matching problem as described below.

3.1. Computing common part of a pair of terms

Through a simple algorithm in Fig. 3, we illustrate how to reduce common part computation of s and t into a string matching problem. s and t are traversed in preorder and stored in arrays S and T respectively. Two pointers, p_s and p_t, are used to scan S and T respectively. Upon termination if *fail* is true then there is no common part and hence unification fails; otherwise the common part is the term obtained by deleting the terms in the frontier. The following theorem from [10] establishes the correctness of the simple algorithm.

Theorem 2. *$S = T$ iff $s = t$.*

3.1.1. Improving running time

Observe that our simple 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 node labels currently being compared are both function symbols 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 labeled with function symbols. Finally, it enters a new skip phase whenever one of the nodes being compared is labeled with a variable. The computations performed in the two phases are as follows. If the pair of function symbols compared in a match phase are identical then p_s and p_t are both incremented by one. A mismatch on the other hand, indicates absence of common part (and hence the failure of unification). For the skip phase, suppose (without loss of generality) p_s points to a node labeled with a variable, say x, and p_t points to some node, say v. Then p_s is advanced by one whereas p_t skips the entire subtree rooted at v and advances to the node immediately following the last node in the subtree rooted at v. Suppose q denotes the subterm rooted at v then the elementary equation $x = q$ is added to the frontier.

Observe the total number of comparisons made in the simple algorithm is linear in the size of the input terms. However, the number of distinct phases the algorithm goes through is proportional to the number of substitutions (i.e., the rhs of elementary equations) computed. Also note that each skip phase can be accomplished in O(1) time by storing preorder in an array and keeping a pointer from each node to the position of the last node (in preorder) in the subtree rooted at this node. Therefore if we can accomplish each match phase also in O(1) time then the running time of our algorithm is proportional to the number of substitutions computed by it. We now examine issues related to improving the running time of our algorithm.

3.1.2. String-matching operations

Observe that during unification common parts of input terms as well as their subterms are computed. We refer to the input terms as the *primary terms* and the subterms of primary terms will be referred to as *secondary terms*. We now identify the string matching questions that arise while computing the common part of two primary terms s and p. Each term is transformed into a set of strings by doing a preorder traversal and removing the variables. Thus $f(a, h(x, b))$ is transformed into fah and b. Each such string from a primary (secondary) term is referred to as *primary* (*secondary*) *string*.

Fig 4 depicts the four kinds of string-matching phases that occur in unification of s and p. Fig 4(a) shows the first match phase (i.e., phase starting at the root), whereas Fig 4(b) shows the last match phase (i.e., phase ending at the last leaf). All other match phases must occur between two skip phases, and are called intermediate match phases. There are two cases to consider for an intermediate match phase, depending

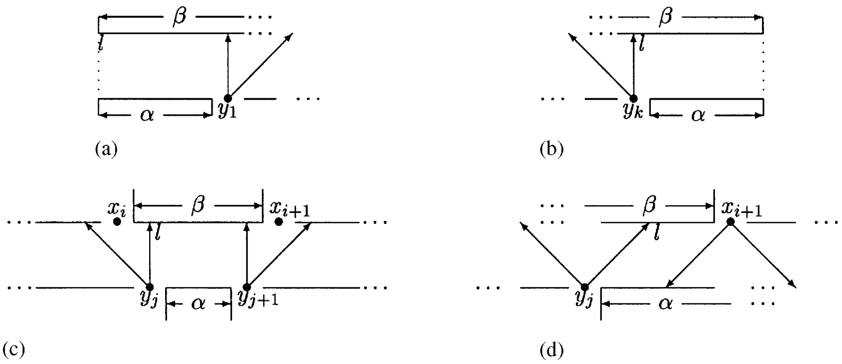


Fig. 4. String matching operations in the match phase.

upon whether the skip phases preceding and following the match phase were (i) both initiated by variables in the same term (Fig 4(c)) or (ii) initiated by variables from different terms (Fig 4(d)). The four scenarios lead to the following string-matching questions.

1. Does α occur at position l in string β (Figs. 4(a)–4(c))
2. Does a given prefix of α occur at position l in β (Fig. 4(d))

Note that both α and β are primary strings and so the above two questions are special cases of the following generic question: Given a specific position l,

Q1 Does a given prefix of a primary string occur in another primary string at l?

We now identify the string matching questions that arise while computing the common part of two secondary terms, say t_1 and t_2. There are three cases.

Case 1: Both t_1 and t_2 have variables. We can again show that a match phase can occur only in the four scenarios shown in Fig. 4. The only difference now is that α and β in Fig. 4(a) denote suffixes of primary strings whereas in Fig. 4(b) they are prefixes of primary strings. Similarly in Fig. 4(d), α can be a prefix of a primary string and β can be a suffix of a primary string. The string matching questions here are:

Q1 (as before) (Figs. 4(b)–4(d))

Q2 Does a given suffix of a primary string occur in another primary string at l? (Fig. 4(a))

Case 2: One of the secondary terms, say t_1, is ground. In this case a match phase can occur only in the three scenarios shown in Fig. 4(a)–4(c) (without x_i and x_{i+1}). (The scenario shown in Fig. 4(d) cannot arise as one of the terms is ground). Herein again α in Fig. 4(a) is a suffix of a primary string whereas it is a prefix of a primary string in Fig. 4(b). Note that in all three scenarios β is a substring of a primary string (representing the preorder of the ground term t_1). It can be easily verified that the generic string matching questions raised here are identical to those in case 1 above.

Case 3: Both t_1 and t_2 are ground. In this case, computing the common part reduces to verifying whether t_1 and t_2 are identical.

In summary, based on the above discussion, the string matching question that arise in any match phase are: Given a specific position l,

Q1 Does a given prefix of a primary string occur in another primary string at l?
Q2 Does a given suffix of a primary string occur in another primary string at l?
Q3 Are preorders of two ground terms equal?

If we can answer any instance of these three questions in O(1) time then each match phase can also be done in O(1) time. **Q3** can be answered by verifying that the two ground terms are identical. Such a verification can be done in O(1) time by assigning an integer signature (varying from 1 to n) to the nodes in the term such that two nodes get the same signature if and only if the subterms rooted at them are identical. Such an encoding can be easily computed by a preprocessing step in time proportional to the size of the term (see [2] for one such method). Upon assigning these signatures we can check whether two ground terms are identical by comparing the signatures assigned to their roots. This comparison takes only a O(1) time and hence **Q3** can be answered in O(1) time. We now show how to answer **Q1** and **Q2** in O(1) time by preprocessing the primary terms s and p into a string matching automaton. Note that the O(1) time taken to answer these questions does not include the preprocessing costs.

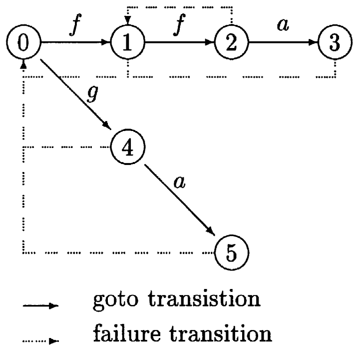
3.1.3. Preprocessing primary terms

Central to our technique is a finite-state automaton that is constructed from the primary 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.

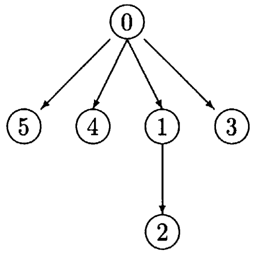
The automaton consists of nodes called *states* and two types of links — *goto* and *failure*. The goto links are labeled with symbols from the alphabet of the keywords. These links together with the states form a "tree-like" structure known as the *goto tree* whose root is the start state (See Fig. 5 for illustration). Following [1] we say state γ *represents* string λ if the path in the goto tree from the start state (the root node) to state γ spells out λ. The construction using 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 text $a_1a_2\ldots a_ja_{j+1}\ldots a_n$. If there is a goto link labeled a_{j+1} from u to w then the automaton makes a goto transition to w. Now,

Lemma 1 (Aho–Corasick). *The string represented by w is the longest suffix of $a_1a_2\ldots a_{j+1}$ that is also a prefix of some keyword.*



(a)



(b)

Fig. 5. Automaton for rule strings $f\,f\,a$, f and ga. (a) Automaton; (b) fail tree.

For example, upon reading $s\,f\,f$ from the string $s\,f\,f\,g\ldots$, the automaton in Fig. 5 is in state 2 which represents $f\,f$. Observe that $f\,f$ is the longest suffix of $s\,f\,f$ that is also the prefix of the keyword $f\,f\,a$.

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:

Lemma 2 (Aho-Corasick). *The string represented by v is longest proper suffix* (*among those represented by the states of the automaton*) *of the string represented by u.*

For the input string $s\,f\,f\,g\ldots$ above, on reading the symbol g in state 2 the automaton makes a transition to state 1. The string f represented by state 1 is the longest proper suffix of $f\,f$, the string represented by state 2.

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. However, recall that we need to know whether a prefix or a suffix of a primary string occurs in another primary string.

We first extend the automaton to handle **Q1**. For clarity of notation we will implicitly assume the presence of position l in every instance of **Q1**. The primary strings of the terms to be unified form the keywords of this automaton. Therefore each prefix of a primary string is represented by state in the automaton.

Suppose we want to know whether a given prefix α of a primary string occurs in another primary string β (see Fig. 6). Observe that γ is a prefix of β. Therefore there is a state s_γ in the automaton that represents γ. Note α is also a prefix of a primary

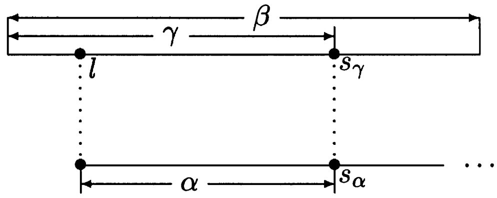


Fig. 6. Answering an instance of **Q1**.

string and hence there is a state s_α that represents α. Now α occurs at l iff α is a suffix of γ. In other words:

Theorem 3. *α is a suffix of γ iff s_α is reachable from s_γ through zero or more failure transitions only.*

The proof of this result is a straightforward consequence of Lemma 2. To handle **Q2**, observe that it is a symmetric dual of **Q1**, i.e., suppose we reverse all the primary strings then a suffix of a primary string is prefix of its reverse. Thus, we can handle **Q2** using an automaton built with the reverses of primary strings.

Now we describe how to answer both these questions in O(1) time. Observe that each state has a unique fail state. So by deleting all the goto transitions and reversing the directions on failure transitions we obtain the *fail tree* of the automaton. (Fig. 5(b) is the fail tree for the automaton in Fig. 5(a).) To each node in this fail tree we assign its preorder number (*pn*) and the number of descendants (*nd*) in its subtree. For α to occur in β at l, s_α must be an ancestor of s_γ in the fail tree (i.e., verify $pn(s_\alpha) \leqslant pn(s_\gamma) \leqslant pn(s_\alpha) + nd(s_\alpha)$). Since this can be verified in O(1) time we can therefore answer **Q1** in O(1) time. Similarly, **Q2** can also be answered in O(1) time using the fail tree of the automaton based on the reverses of primary strings.

3.1.4. Algorithmic details

Based on the discussions in the previous section we now present the details of procedure *CommonPart* that computes common part based on string-matching automaton. The two primary terms are preprocessed to construct the two Aho–Corasick automata (one for the primary strings and another for their reverses) and their fail trees. We use the following data structure to represent the preorders of primary terms and their subterms. We use two arrays P and S to store the preorders of the two primary terms p and s. Given the array P (or S), the preorder of any subterm t of p (or s) can be specified by giving the two endpoints of the preorder of t in P (or S). Therefore we represent preorder of any term t by the triple $\langle (X, i, j) \rangle$ where X is the preorder of a primary term and i and j mark the two endpoints of preorder of t in X.

A record in arrays P and S has six fields: *label*, *subtree*, *varposn*, *code*, *state* and *revstate*. The *label* fields are used to store the labels of nodes that appear in preorder. The *varposn* field in $P[i]$ is set to the preorder number of the nearest variable node that appears after i in preorder. The *subtree* field of $P[i]$ is set to j if $P[j]$ contains information about the last node (in preorder) in the subtree rooted at the node specified in $P[i]$. The *code* field is set to the code obtained by precessing the terms using the congruence closure method [2]. This field is used only for comparing ground subterms. The *state* field specifies the state of the automaton (built with the primary strings) reached on reading $P[i].label$ while scanning P. Similarly, *revstate* specifies the corresponding state reached on scanning the reverse of P with the automaton built with reverses of the primary strings. The structure of array S is identical to P. In addition to these arrays *CommonPart* uses local variables p_1, p_2, l_1, l_2 and *lastvar*. p_1 and p_2 point to positions in the preorders of the two input terms upto which *CommonPart* has proceeded without failure. l_1 and l_2 are the lengths of remaining portions of two primary strings (in the input terms) from p_1 and p_2 respectively. *lastvar* is set to q (q is either 1 or 2) if the immediately preceding substitution was made to a variable in the qth input term. *pn* and *nd* are functions that return the preorder number and the number of descendants of a state (or revstate) in appropriate fail tree.

The common part of two terms t_1 and t_2 is computed by invoking *CommonPart* (t_1, t_2). In the description below we use function $pre(t)$ to retrieve the triple denoting the preorder of t.

Procedure $CommonPart(t_1, t_2)$
begin
$fail :=$ **false**;
Let $\langle T_1, s_1, e_1 \rangle = pre(t_1)$;
Let $\langle T_2, s_2, e_2 \rangle = pre(t_2)$;
$p_1 := s_1$; $p_2 = s_2$; {initialize the pointers}
Let $Frontier =$ **nil** {and Frontier to empty list}
{Check whether both are ground terms}
if $(T_1[p_1].varposn > e_1)$ **and**$(T_2[p_2].varposn > e_2)$**then**
  return $(T_1[p_1].code \neq T_2[p_2].code)$;
endif;
$l_1 := \min(T_1[p_1].varposn, e_1) - p_1 + 1$; {length of the first string in first term}
$l_2 := \min(T_2[p_2].varposn, e_2) - p_2 + 1$; {length of the first string in second term}
{First phase is always a match phase. So, perform string match.}
$pn_1 := pn(T_1[p_1].revstate)$;
$pn_2 := pn(T_2[p_2].revstate)$;
$nd_1 := nd(T_1[p_1].revstate)$;
$nd_2 := nd(T_2[p_2].revstate)$;
if $l_1 < l_2$ **then**
  {String in the first term is shorter}
  $fail := \neg(pn_1 \leqslant pn_2 \leqslant pn_1 + nd_1)$;
  $p_1 := p_1 + l_1$;
  $p_2 := p_2 + l_1$;

else
  {String in the second term is shorter or equal}
  $fail := \neg(pn_2 \leqslant pn_1 \leqslant pn_2 + nd_2)$;
  $p_1 := p_1 + l_2$;
  $p_2 := p_2 + l_2$;
endif
while$\neg fail$ **and** $p_1 < e_1$ **and** $p_2 < e_2$ **do**
  {Perform substitutions as long as one of the term has a variable}
  while $\mathbf{T_1[p_1]}$.label or $\mathbf{T_2[p_2]}$.label is a variable do
    if $T_1[p_1].label$ is a variable then
    {Compute substitution for variable at $T_1[p_1]$}
    $lastvar := 1$;
    $Frontier := append(\text{“}T_1[p_1].label = \langle T_2, p_2, T_2[p_2].subtree\rangle\text{”}, Frontier)$
    $p_1 := p_1 + 1$;
    $p_2 := T_2[p_2].subtree + 1$;
else
  {Compute substitution for variable at $T_2[p_2]$}
  $lastvar := 2$;
  $Frontier := append(\text{“}T_2[p_2].label = \langle T_1, p_1, T_1[p_1].subtree\rangle\text{”}, Frontier)$
  $p_1 := T_1[p_1].subtree + 1$;
  $p_2 := p_2 + 1$;
endif
endwhile
{If we have not reached the end of the terms' preorders then we
have to perform a string match as both p_1 and p_2 point to functor nodes}
if $p_2 < e_2$ **and** $p_1 < e_1$ **then**
  $l_1 := \min(T_1[p_1].varposn, e_1) - p_1 + 1$;
{length of current string in first term }
  $l_2 := \min(T_2[p_2].varposn, e_2) - p_2; +1$
{length of current string in second term }
  if$l_1 < l_2$ **then**
    { String in the first term is shorter }
    $pn_1 := pn(T_1[p_1 + l_1 - 1].state)$;
    $pn_2 := pn(T_2[p_2 + l_1 - 1].state)$;
    $nd_1 := nd(T_1[p_1 + l_1 - 1].state)$;
    $nd_2 := nd(T_2[p_2 + l_1 - 1].state)$;
    $p_1 := p_1 + l_1$;
    $p_2 := p_2 + l_1$;
  else
    {String in the second term is shorter}
    $pn_1 := pn(T_1[p_1 + l_2 - 1].state)$;
    $pn_2 := pn(T_2[p_2 + l_2 - 1].state)$;
    $nd_1 := nd(T_1[p_1 + l_2 - 1].state)$;

```
63.         nd2 := nd(T2[p2 + l2 − 1].state);
64.         p1 := p1 + l2;
65.         p2 := p2 + l2;
66.     endif
67.     if lastvar = 1 then
68.         fail := ¬(pn1 ⩽ pn2 ⩽ pn1 + nd1);
69.     else
70.         fail := ¬(pn2 ⩽ pn1 ⩽ pn2 + nd2);
71.     endif
72. endif
73. endwhile
74. if ¬fail then return(Frontier)
end
```

Suppose m is the total number of substitutions computed in *CommonPart*. Then,

Theorem 4. *CommonPart takes* $O(m)$ *time.*

Proof. Since lines 1–27 take only $O(1)$ time the complexity of *CommonPart* is given by the time taken to execute the outer while loop (lines 28–73). Lines 45–73 in the outer while loop take only $O(1)$ time. Furthermore, for each iteration of the outer loop, the inner loop is executed at least once. Therefore, the time taken to execute the inner while loop over all iterations of the outer while loop will dominate the time complexity. Since each iteration of inner while loop takes $O(1)$ time and computes one substitution, the total time taken by *CommonPart* is $O(m)$. □

Let k be the total number of *occurrences* of variables in terms s and t. As an immediate consequence of the above theorem:

Corollary 1. *CommonPart*(s,t) *requires at most* $O(k)$ *time.*

In an equation queue E_x there can be several terms. Procedure *CommonPart* computes the common part of a pair of terms. It can be extended to compute the common part of several terms together. For example, suppose E_x consists of $f(x_1), f(f(x_2))$, $f(f(f(x_3)))$. Observe that a set of elementary equations equivalent to E_x can be generated by first computing the common part of $f(x_1)$ and $f(f(x_2))$ followed by the common part of $f(f(x_2))$ and $f(f(f(x_3)))$ and combining the frontiers together. We now formalize the *bidirectional-reduce* (BR) operation described above as follows. Let $t_1, t_2, \ldots, t_n$ be the terms in E_x. To reduce them to an equivalent set of elementary equations we invoke procedure common part $n-1$ times. In the ith application we compute the common part of t_i and t_{i+1}.

Note that it is also possible to reduce E_x by computing common part of t_1 with every t_i $(2 \leqslant i \leqslant n)$, a pair at a time. In this case t_1 is used several times. In contrast, the BR operation uses a term at most twice. We show later on that this property of the BR operation yields good performance in important cases of subterm unification.

4. Analysis of automata-driven unification

We now analyze the complexity of our unifiction algorithm based on string-matching automaton. The analysis is split into two cases based on the structure of s and t, namely, (1) only one is linear and (2) both are nonlinear. For the case where both are linear we will show that is a special case of (1). Our analysis exploits the fact that in most applications requiring subterm unification such as critical pairs, paramodulation and narrowing, s and p do not share any variables. Even otherwise it is possible to encode them into two other terms that do not share variables with an increase only by a constant factor in the size of s and p and the number of variables in them. In the remainder of this section we use t to denote any subterm of p and k to denote the total number of variable occurrences in both s and t together.

4.1. Linear–nonlinear unification

Without loss of generality let s be linear and t be nonlinear. Let $E_{x_1}, E_{x_2}, \ldots, E_{x_n}$ be the collection of nonempty equation queues obtained by invoking procedure *CommonPart* on s and t (Note x_i is a variable whose equation queue is E_{x_i}.) Without loss of generality, let $x_1, x_2, \ldots, x_m$ be the variables in t and $x_{m+1}, \ldots, x_n$ be those in s.

Note E_{x_i} denotes a set of simultaneous equations of the form $x_i = t_1$, $x_i = t_2, \ldots,$ $x_i = t_l$ where t_j is a term in E_{x_i}. Let $Sol(E_{x_i})$ denote the canonical set of equations equivalent to E_{x_i}. We now show that solution to $s = t$ can be obtained by computing each $Sol(E_{x_i})$ independently and appending them together.

Observe that $x_{m+1}, x_{m+2}, \ldots, x_n$ are the variables from the linear term s. Recall that s and t do not have any variables in common. This means each E_{x_i} $(m < i \leqslant n)$ contain only one term and that term is a subterm of t. Furthermore x_i $(m < i \leqslant n)$ cannot occur in any substitution. Therefore, the collection of equations in $E_{x_{m+1}}, E_{x_{m+2}}, \ldots, E_{x_n}$ are already in canonical form. Let Π be this collection. Observe that $Sol(E_{x_i})$ $(1 < i \leqslant m)$ denotes the solution obtained by solving equations in E_{x_i} in isolation. It can be easily shown that:

Lemma 3. *$\bigcup_{i=1}^{j} Sol(E_{x_i})$ $(j \leqslant m)$ is in canonical form.*

Lemma 4. *Let $\Gamma = \bigcup_{i=1}^{m} Sol(E_{x_i})$. The sequence of equations obtained by appending Γ to Π is in canonical form.*

The above lemma implies that each E_{x_i} can be processed in isolation and the solutions appended together is the solution for $s = t$. Let V_i be the set of variables occurring in the terms in E_{x_i}.

Lemma 5. *In computing $Sol(E_{x_i})$ we make at most* $\mathrm{O}(|V_i|)$ *substitutions.*

Proof. Note that substitutions are made only in procedure *CommonPart*. We use a counting technique to prove this result. In this counting technique we associate an *occurrence counter* and *substitution counter* with each variable. The substitution counter of a variable keeps count of the total number of substitutions made to that variable. We say that a term in an equation queue is processed iff procedure *CommonPart* will never be invoked on it. All other terms are set to be unprocessed. An occurrence counter of a variable, say x, keeps a count of the number of occurrences of x in all unprocessed terms. Intuitively, this count is an upper bound on the number of substitutions x can take in common part computations yet to be done on the unprocessed terms. Recall that reducing an equation queue involves iterating over three major steps, viz: (1) BR operation (2) *MergeQ* and (3) *UpdateCounters* (see while loop in procedure *Unif* in Fig. 2). We use induction on number of such iterations.

Let $t_1, t_2, \ldots, t_l$ be the terms in E_{x_i}. The occurrence counter of each variable in t_1 and t_l is initialized to 1 whereas the occurrence counter of the variables in all other terms is set to 2. This is because in the BR operations each of these terms is used twice. To begin with the substitution counter of each variable is initialized to 0. We now show by induction that the sum of the two counters is at most two.

Base case: Since we use BR operation, we invoke procedure *CommonPart* $l-1$ times. In these invocations t_j is used at most twice-once in *CommonPart*(t_{j-1}, t_j) and again in *CommonPart*(t_j, t_{j+1}). Let x be a variable in t_j. Suppose we compute a substitution for x in *CommonPart*(t_{j-1}, t_j) then x cannot be in any subterm computed as substitution for any variable in t_{j-1} (terms in E_{x_i} are nonoverlapping subterms of s). The same argument holds for *CommonPart*(t_j, t_{j+1}). Therefore if two substitutions are computed for x then it cannot appear in the rhs of any equations appearing in the frontier computed in the two invocations of *CommonPart*. As x cannot appear on any other term in E_{x_i}, x cannot appear on the rhs of equations appearing in the frontier computed in other $l-3$ invocations of *CommonPart* also. This means that at the end of this iteration the occurrence counter of x becomes 0. Therefore the sum of the counter remains two. On the other hand, suppose x takes a substitution only in one invocation, say *CommonPart*(t_{j-1}, t_j), and not in the other. Clearly in *CommonPart*(t_j, t_{j+1}) a substitution containing x must have been computed for a variable, say y. Now at the end of this iteration E_y will have a term containing an occurrence of x. Once again using the fact that x occurs only in t_j we can show that this will be the only occurrence of x among the terms in equation queues remaining at the end of this iteration. Therefore in this case also the sum of the counters is at most 2 at the end of the iteration. Finally in the case when x does not acquire any substitution in both invocations of *CommonPart* involving t_j, it can be shown by similar arguments that the sum of the two counters is at most two.

Induction step: Assume the claim is true at the end of qth iteration. Let E_y be the equation queue processed in the $(q+1)$th iteration. Since the substitution counter of y is at most two at the end of qth iteration, E_y has at most two terms. Furthermore, by induction hypotheses each variable can occur at most two times among the terms in E_y. Since E_y has at most two terms, there will be only one invocation of *CommonPart* and the two terms in E_y will be used only once. By arguments similar to those used in the base case we can again show that at the end of this iteration the sum of the counters of all variable remains at most two. □

We remark that the proof of the induction step in the above lemma crucially depends on the fact that each term is used at most once. This is a direct consequence of applying the BR operation in the first iteration. Had we used each term more than two times in the first iteration we cannot obtain this bound. Since the complexity of procedure *CommonPart* is proportional to the number of substitutions computed in it (see Theorem 4), T_{cp} for computing $Sol(E_{x_i})$ is $O(|V_i|)$. We now show that T_{merge} and T_{update} are also $O(|V_i|)$ in computing $Sol(E_{x_i})$.

Corollary 2. *T_{update} is $O(|V_i|)$ for computing $Sol(E_{x_i})$.*

Proof. The technique of updating occurrence counter used in the proof of Lemma 5 can be used to implement procedure *UpdateCounter*. Hence the result. □

While solving E_{x_i} we compute at most two substitutions for each variable. We can exploit this fact to implement *MergeQ* efficiently. Note that *MergeQ* forms a single equation queue from all those merged together. In addition, a leader is chosen from the variables whose equation queues have been merged. A substitution computed (later) for any variable in this merged group is placed in the leader's equation queue. Therefore, all the variables in the merged group must know their current leader. In general, a merged group can grow dynamically requiring constant update of leader information. Suppose it can be guaranteed that a variable is not going to acquire any new substitution then its leader information need not be updated when the group expands. We use this fact to implement *MergeQ* efficiently. The details are as follows. Let $x_1, x_2, \ldots, x_l$ be the variables in a merged group.

Lemma 6. *There are at most two variables that can acquire new substitutions.*

Proof. Construct a graph G as follows. The nodes of G are $x_1, x_2, \ldots, x_l$. There is an edge between x_i and x_j iff the $x_i = x_j$ was generated. $x_1, x_2, \ldots, x_l$ constitutes one group and so G must be connected. Observe that when $x_i = x_j$ is generated both x_i and x_j acquire a substitution each. Since there can be at most two substitutions for any variable, G must be a chain. Obviously, the two variables at the ends of the chain can alone acquire any additional substitutions. □

Lemma 7. *Each invocation of MergeQ takes only* $O(1)$ *time.*

Proof. Let L_1 and L_2 be the two groups to be merged. Let $x_1, x_2, \ldots, x_n$ be the variables in L_1 and $y_1, y_2, \ldots, y_m$ be the variables in L_2. Without loss of generality let $x_1 = y_1$ be the equation that causes L_1 and L_2 to be merged. Arbitrarily pick the leader of L_1

to be the leader of the merged group. Since x_n and y_n alone can acquire any new substitutions, we update only their leader information. Hence the lemma. □

Lemma 8. *T_{merge} is* $O(|V_i|)$ *in computing* $Sol(E_{x_i})$.

Proof. By the previous lemma each invocation of *MergeQ* takes only O(1) time. As there are at most $O(|V_i|)$ invocations of *MergeQ*, T_{merge} is $O(|V_i|)$. □

Since each of T_{cp}, T_{update} and T_{merge} is $O(|V_i|)$:

Corollary 3. *Computing* $Sol(E_{x_i})$ *takes* $O(|V_i|)$ *time.*

Combining the above results we get:

Theorem 5. *Unification of s and t takes* $O(k)$ *time.*

Proof. From Lemma 4, solution to $s = t$ is obtained by computing $Sol(E_{x_i})$ $(1 \leqslant i \leqslant n)$ in isolation and appending them together. From the above corollary computing $Sol(E_{x_i})$ takes $O(|V_i|)$ time. Furthermore $\sum_{i=1}^{n} |V_i| \leqslant k$. Hence the theorem. □

Observe that the above theorem also holds for the special case when both s and t are linear.

4.2. Nonlinear–nonlinear unification

We now analyze the case where both s and t are nonlinear. Unlike the previous case, the equation queues in this case do not have any special structure. Specifically, each equation queue now can have many terms and the substitutions computed can be overlapping subterms. Therefore the variable occurrences in them can get duplicated arbitrarily as shown below.

Consider the scenario (see Fig. 7) when invoking procedure *CommonPart* first on the pair of terms t_{i-1} and t_i and again on pair t_i and t_{i+1}. Let t_1 and t_2 be the substitution computed for x_r and x_q as a result of these invocations. Observe that y occurs both in t_1 and t_2. So what was one occurrence of y in t_i has now become two distinct occurrences in terms t_1 and t_2. We refer to this as *occurrence duplication* of y. It is possible for t_1 and t_2 to be used in common part computations later on and they in turn can duplicate occurrences of y further. Some (or all) the new occurrences of y can acquire substitutions. So occurrence duplication can adversely affect the complexity of our method. Suppose k_d is the number of distinct variables in s and t together then (with occurrence duplication) it is trivial to establish an upper bound of $O(k2^{k_d})$ substitutions that can be potentially computed when unifying s and t. But we now establish a tighter bound of $O(k_d k)$.

Lemma 9. *At most* $O(k_d k)$ *substitutions are computed in the unification of s and t.*

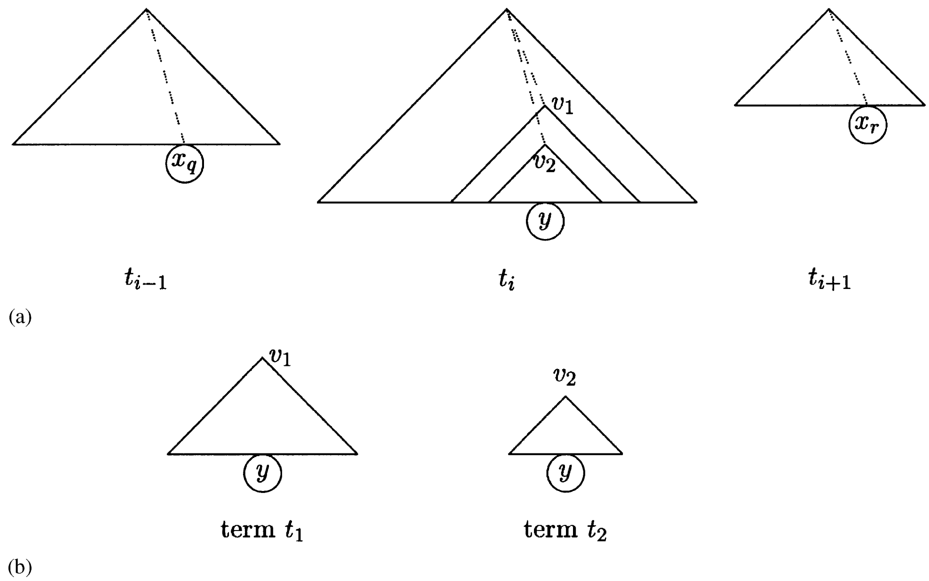
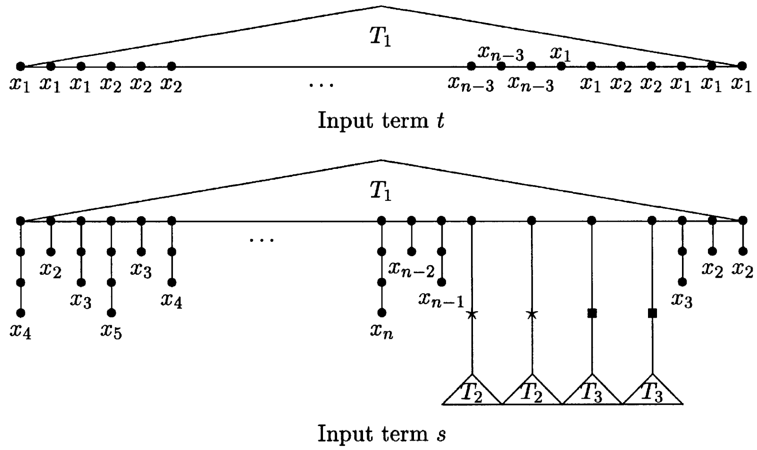


Fig. 7. Occurrence duplication of variables. (a) Two *CommonPart* invocations involving t_i; (b) substitutions for x_q and x_r.

Proof. Let $t_1, t_2, \ldots, t_l$ be the terms in E_x. Now consider the iteration in procedure *Unif* when E_x is processed. Observe that for s and t to unify the terms in E_x must also unify. This means that there cannot be two terms t_i and t_j in E_x such that t_i is a subterm of t_j (we deal with finite terms only). Therefore, the terms in E_x must be nonoverlapping subterms of s and t. This means that there can be at most k occurrences of variables among the terms in E_x. In case we compute $2k+1$ substitutions in this iteration then there are two terms in E_x such that one is subterm of another and hence unification fails. Hence the number of substitutions computed in this iteration is at most $O(k)$. Note that this argument holds for any iteration. Since there are k_d distinct variables there can be at most k_d iterations and hence the bound. $\square$

Observe that k_d and k can both be $O(|s|+|t|)$ and hence the unification can become quadratic. In fact, we show that this bound is tight through a carefully constructed nontrivial example in Fig. 8. Now the interesting question is whether this bound can be improved when no variables in s and t occur more than q times for a fixed q. It can be shown that any such q-occurrence case can be converted into a 3-occurrence case with an increase only by a constant factor in $|s|+|t|$ and k. This means the above bound is tight for any n-occurrence case for $n \geqslant 3$. However, for $n = 2$, we can improve the bound to $O(k)$. This is because each equation queue will contain only two terms and hence the arguments used in (induction step of) the proof of Lemma 5 apply, i.e., the sum of the occurrence and substitution counters will never be more than 2 and hence we have:



- All nonleaf nodes in s and t are labeled by either f or g. f and g have arities 2 and 1 respectively.
- T_1, T_2 and T_3 represent portions of s and t that are full binary trees.
- T_2 and T_3 have $O(n)$ leaves that are labeled by the variable x_{n+1}.
- The paths in s marked by ★ and ■ contain $n-1$ and $n-2$ nodes (labeled by g) respectively.

The equation queues computed in the unification of s and t:

$$E_{x_1} = \{g^3(x_4), g(x_2), g^2(x_3), g^{n-1}(T_2), g^{n-1}(T_2), g^2(x_3), g(x_2), g(x_2)\}$$
$$E_{x_2} = \{g^3(x_5), g(x_3), g^2(x_4), g^{n-2}(T_3), g^{n-2}(T_3), g^2(x_4), g(x_3), g(x_3)\}$$
$$E_{x_3} = \{g^3(x_6), g(x_4), g^2(x_5), g^{n-3}(T_2), g^{n-3}(T_2), g^2(x_5), g(x_4), g(x_4)\}$$
$$E_{x_4} = \{g^3(x_7), g(x_5), g^2(x_6), g^{n-4}(T_3), g^{n-4}(T_3), g^2(x_6), g(x_5), g(x_5)\}$$
$$\vdots \qquad\qquad\qquad\qquad \vdots$$
$$E_{x_{n-3}} = \{g^3(x_n), g(x_{n-2}), g^2(x_{n-1}), g^3(T_2), g^3(T_2), g^2(x_{n-1}), g(x_{n-2}), g(x_{n-2})\}$$
$$E_{x_{n-2}} = \{g^2(T_3), g^2(T_3), g^2(x_n), g(x_{n-1}), g(x_{n-1})\}$$
$$E_{x_{n-1}} = \{g(T_2), g(T_2), g(x_n)\}$$
$$E_{x_n} = \{T_3, T_2\}$$

Fig. 8. Example in which $O(n^2)$ substitutions are computed.

Theorem 6 (2-occurrence nonlinear unification). *When no variable in either s or t occurs more than twice then unification of s and t requires at most* $O(k)$ *time.*

Proof. As explained above the argument used in the proof of Lemma 5 carries over in this case. Hence we can again show that each variable acquires at most two substitution. Consequently T_{cp}, T_{update} and T_{merge} are $O(k)$ and hence unification takes $O(k)$ time. □

5. Improving the efficiency of nonlinear–nonlinear unification

We now describe modifications to our algorithm so that its complexity is at most linear for q-occurrence ($q>2$) cases. Specifically, these modifications guarantee that when s and t have arbitrary number of variable occurrences, we compute at most $O(\min\{k_d k, |s|+|t|\})$ substitutions. The key idea here is to prevent occurrence duplication through sharing. Consider Fig. 7 where the two substitutions computed represent the equations $x_q = t_1$ and $x_r = t_2$. Now suppose $\overline{t_1}$ is a term obtained by replacing the subterm t_2 in t_1 by x_r. Clearly $x_r = t_2$ implies that $t_1 = \overline{t_1}$. In other words, the set of equations $\{x_q = t_1, x_r = t_2\}$ is equivalent to $\{x_q = \overline{t_1}, x_r = t_2\}$. This means that the solution to the unification problem and hence the correctness of the algorithm will not change if we use $\overline{t_1}$ instead of t_1. Furthermore, observe that the occurrence of y is not duplicated if $\overline{t_1}$ is used as a substitution for x_q instead of t_1. However now, $\overline{t_1}$ instead t_1 must participate in common part computations requiring addition of new strings to the automaton. However explicit addition of new strings (and hence explicit creation of $\overline{t_1}$) defeats the whole purpose of preprocessing. Hence we use a *marking* technique to *simulate* $\overline{t_1}$ without modifying t_1 explicitly. The details are as follows.

We say that node v is a *variable node* if it is labeled by a variable. We say that v is the *first variable node* in term t iff its preorder number is the smallest among the variable nodes in t.

Let P_1 and P_2 denote the preorders of t_1 and t_2 (see Fig. 9). Let $\overline{P_1}$ denote the preorder of $\overline{t_1}$. Suppose we place a mark on v_2 in P_1. By using this mark we can simulate $\overline{P_1}$. The mark also represents variable x_r in $\overline{P_1}$. To compute common part of $\overline{P_1}$ with any other term, say P', *CommonPart* will perform string-matching operations

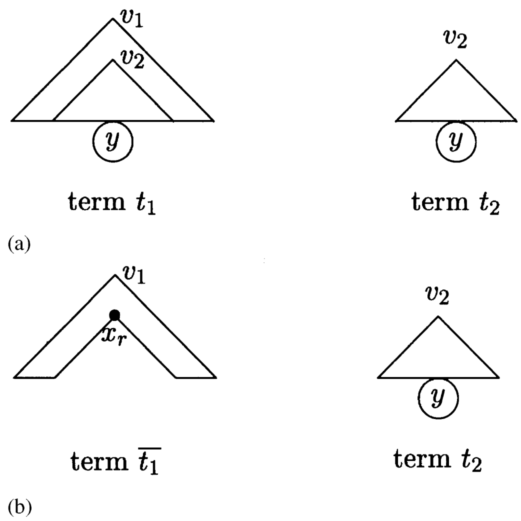


Fig. 9. Avoiding occurrence-duplication. (a) Substitutions for x_q and x_r; (b) modified Substitutions for x_q and x_r.

involving preorder strings of $\overline{P_1}$ and P'. This means the preorder strings of P_1 along with the mark placed at v_2 must be used in place of the preorder strings of $\overline{P_1}$. Now recall that prior to performing a string-matching operation, procedure *CommonPart* first computes the length of the strings involved in a match (see lines 10, 11 and 48–49). This is done using the position of the variables in the preorder of the term. Therefore we must know the position of the mark in P_1 in order to simulate the preorder strings in $\overline{P_1}$. If we place the mark on v_2 then we must scan P_1 one symbol at a time to determine its position. But doing so will degrade performance. Furthermore, it runs counter to our objective of not inspecting symbols one at a time. Therefore, we physically place the mark on the first variable, say y in P_2. With this mark we also maintain information about the two endpoints of P_2 in P_1. By retrieving this information at y we can compute the lengths of preorder strings of $\overline{P_1}$. If t_1 is a ground term then no occurrence duplication is possible and hence there is no need for a mark.

We now define a mark formally. Recall that we store preorders of primary terms in an array and specify preorders of subterms using a triple notation (see Section 3.1.4). Let t be a substitution for x. Suppose i and j are the two endpoints of preorder of t and w is the root of t then:

Definition 1 (*Mark*). A mark associated with the node w is the triple $\langle i,j,x\rangle$ and it is physically placed on the first variable node in t (i.e., the subterm rooted at w).

We say that the node w is the vertex associated with the mark $\langle i,j,x\rangle$. We can view this mark as creating a *virtual variable* x on w and we say that w is labeled by this virtual variable x. This virtual variable is like any other occurrence of x, i.e., it can acquire substitutions which will be placed in E_x and it can also trigger a *MergeQ* operation.

Let $M_1, M_2, \dots, M_n$ be the marks (physically) placed on a variable node in term t. Let $v_1, v_2, \dots, v_n$ be the vertices associated with $M_1, M_2, \dots, M_n$ respectively. It is quite easy to see that for any pair v_i, v_j either v_i is an ancestor of v_j or vice versa.

Definition 2. M_i is said to be the outermost mark with respect to t iff v_i is the closest descendant (among v_j's $1 \leqslant j \leqslant n$) of the root of t.

Next we show how the marks are used to obtain the simulated terms and their preorder. Given a term t with marks placed on its variables the simulated term $\bar{t}$ is obtained as follows. For each outermost mark $\langle i,j,x\rangle$ associated with a node v replace subterm rooted at v by the variable x. The resulting term is $\bar{t}$. For example, consider the subterm t of s in Fig. 10. Here m_1 and m_2 are the only two outermost marks in t. We obtain $\bar{t}$ by replacing the subterms t_1 and t_2 in t by the variables x_1 and x_2 respectively. Suppose $pre(q)$ denote the preorder of term q then we now show how to obtain $pre(\bar{t})$ from $pre(t)$. Observe that $pre(t)$ is represented by the triple $\langle S, begin, end\rangle$ where S is preorder of s, and *begin* and *end* are two endpoints of preorder of t in S. Now $pre(\bar{t})$ is $\alpha_1 x_1 \alpha_2 x_2 \alpha_3$ where $\alpha_1 = \langle S, begin, i_1\rangle$, $\alpha_2 = \langle S, j_1, i_2\rangle$ and $\alpha_3 = \langle S, j_2, end\rangle$. In

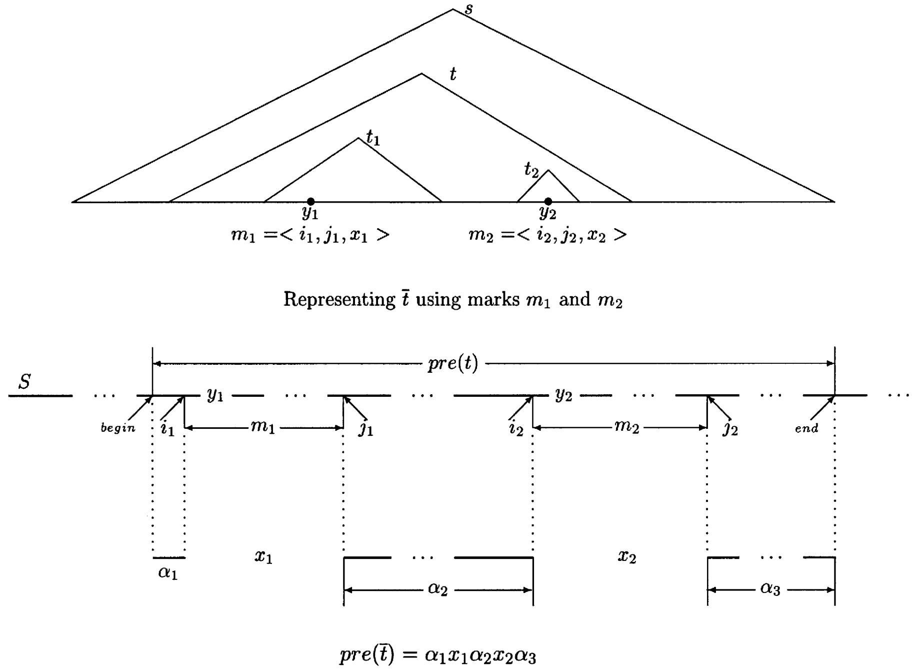


Fig. 10. Extracting simulated term $\bar{t}$, and its preorder $pre(\bar{t})$.

general, if the set of outermost marks are $\{\langle i_1, j_1, x_1\rangle, \langle i_2, j_2, x_2\rangle, \ldots, \langle i_n, j_n, x_n\rangle\}$ then $pre(\bar{t}) = \alpha_1 x_1 \alpha_2 x_2 \ldots \alpha_n x_n \alpha_{q+1}$ where $\alpha_1 = \langle S, begin, i_1\rangle$, $\alpha_{q+1} = \langle S, j_n, end\rangle$ and $\alpha_l = \langle S, j_{l-1}, i_l\rangle$ $(2 \leqslant l \leqslant n)$.

A mark is placed every time a nonground and non-variable term is computed as a substitution. Observe that terms in an equation queue E_x are substitutions acquired by x. This means that the root of each nonground term in E_x must be associated with a mark. At the beginning of the iteration (in procedure *Unif*) that processes E_x, these marks are deleted prior to invoking *CommonPart* for the first time (in this iteration). We now show that no unprocessed term is simulated by the deleted marks.

Lemma 10. *If a mark is deleted then it is not the outermost mark for any unprocessed term.*

Proof. Suppose we delete a mark $\langle i, j, x\rangle$ and it is one of the outermost marks for some unprocessed term q. This mark denotes a virtual variable x in q. This means there is an unprocessed term containing x. Hence, E_x could not have been selected for processing and so this mark could not have been deleted. $\square$

To quickly access the outermost mark the set of marks on a variable node are maintained in a sorted order; sorted in decreasing distance from the variable node. We now show that a stack data structure suffices to maintain this sorted order.

Lemma 11. *If CommonPart is invoked on a term t then the first mark in the sorted list of marks* (*placed on a variable in t*) *is an outermost mark with respect to t.*

Proof. Suppose it is not. Then the node associated with the first mark must be an ancestor of the root of t. In that case we cannot process the equation queue containing t and hence *CommonPart* cannot be invoked with t – a contradiction. Hence the lemma. □

Lemma 12. *When a mark is to be deleted it is the first mark in the sorted list.*

Proof. Recall that we delete marks only at the beginning of each iteration (in *Unif*). Suppose E_x is the equation queue being processed. Let t be a term in E_x and m be the mark associated with the root of t. Now suppose $n \neq m$ is the first mark in the sorted list. Let v be the node associated with n and t' be the term rooted at v. Since t' contains an occurrence of x the equation queue containing t' must have been processed earlier. In such a case n must have been deleted – a contradiction. Hence the lemma. □

Lemma 13. *Whenever a new mark is placed on a variable node then its distance is larger than that of any other mark placed on the same node.*

Proof. Let v denote a variable node. Now a new mark can be placed on v only when *CommonPart* is invoked on a term t containing v. Suppose such an invocation computes a substitution that causes a new mark to be placed on v. Let w be the root of the term computed as the substitution. By Lemma 11 the first mark in the sorted list is the outermost mark with respect to t. Obviously, w must be ancestor of the vertex associated with this outermost mark. Hence this new mark must be first in the updated sorted list. □

Based on the above results we show:

Theorem 7. *The sorted list of marks can be organized as a stack.*

Proof. From the above three lemmas it is clear that the marks are inserted, accessed and deleted only from one end of the sorted list. Hence the theorem. □

An immediate consequence of the above result is:

Corollary 4. *A mark can be inserted, accessed and deleted from the sorted list in* O(1) *time.*

5.1. Computing common part of simulated terms

We describe modifications to procedure *CommonPart* to deal with simulated terms. Suppose we want to compute the common part of $\overline{t_1}$ and $\overline{t_2}$. We now show how to compute it using $pre(t_1)$ and $pre(t_2)$ only. Observe that a mark denotes a virtual variable in a simulated term. A variable of a primary term within a substitution is said to be *covered* by the virtual variable corresponding to the substitution. We refer to the variables in the primary terms as *actual variables*. The only variables appearing in a simulated terms are either virtual variables (corresponding to outermost marks) or actual variables not covered by them. In the absence of any virtual variables the string-matching questions would be identical to those discussed in Section 3.1.2. But the presence of virtual variables introduce a new string-matching question. In a simulated term the string appearing between two consecutive virtual variables corresponds to a substring of a primary string (such as α_2 in Fig. 10). To perform a string matching operation with such a string we must now answer whether a substring of a primary string occurs in another at position l. Our automaton cannot directly answer this question. However, we now show that this question can be transformed into an equivalent one that can be answered by the automaton.

The main idea is based on the following observation. $\overline{t_1}$ and $\overline{t_2}$ are created only to avoid occurrence duplication. The solution to unification will remain unaltered had we allowed occurrence duplication, i.e., the equation $t_1 = t_2$ must have a solution for unification to succeed. This means that the common part between t_1 and t_2 must exist. In other words, while computing common part of $\overline{t_1}$ and $\overline{t_2}$ if we can infer that t_1 and t_2 do not have a common part then we can conclude that the common part computation of $\overline{t_1}$ and $\overline{t_2}$ fails regardless of whether common part of $\overline{t_1}$ and $\overline{t_2}$ exists. Based on this observation we can transform the new string-matching questions into those involving strings in $pre(t_1)$ and $pre(t_2)$.

Recall from Fig. 4, the four scenarios that arise in common part computation. Since we are dealing with simulated terms, the variables in Fig. 4 can now be virtual variables. Fig. 11 replicates situations in Fig. 4 for virtual variables. We now show how to transform the string matching questions arising in situations shown in Fig. 11 into those that can be answered by the automaton. We begin with Fig. 11(a). Herein we want to verify whether γ occurs in δ at l. Here, v_1 is a virtual variable in $\overline{t_2}$. Let the mark corresponding to v_1 be placed on the node labeled by variable y_1. Without loss of generality assume $|\alpha| \leqslant |\beta|$. For the common part of t_1 and t_2 to exist α must occur in β at l. As α is a suffix of a primary string (since y_1 is a actual variable) this can be verified in O(1) time by our automaton. Furthermore, if α occurs in β at l then γ also occurs in δ at l. If α does not occur at l in β then unification fails and hence we conclude that the common part computation of $\overline{t_1}$ and $\overline{t_2}$ also fails. Similarly, the string matching questions that arise in the other situations can also be transformed into those than can be answered by the automaton. Specifically, in each of the other cases, we transform the question that checks for occurrence of γ in δ at l into the one that checks for occurrence of α in β at l. Since the transformed questions can be answered by our automaton, we conclude that all the string matching questions that arise in computing the common part of two simulated terms takes only O(1) time.

Note that we do not place marks when ground terms are computed as substitutions. Recall that computing common part of ground terms involves comparing their codes

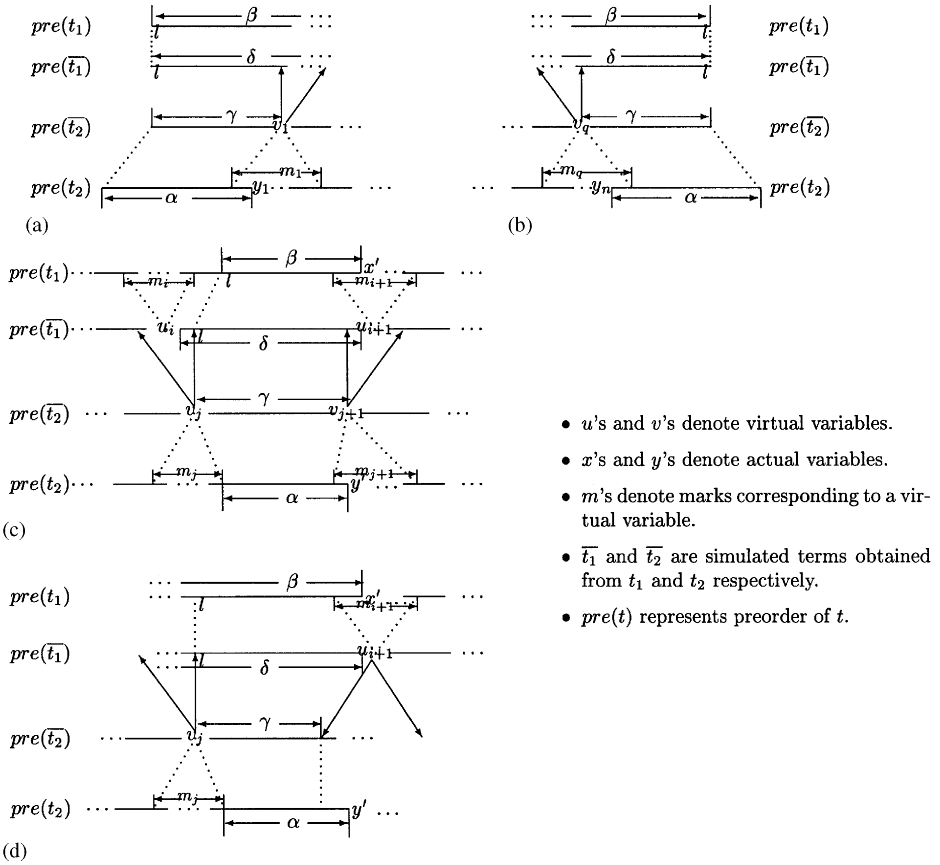


Fig. 11. String matching operations with simulated terms.

only (see Section 3.1.2). Hence, we do not invoke *CommonPart* on two ground terms. Instead, we place all ground terms at the front of the equation queue and compare their codes prior to invoking BR operation. If the codes of these ground terms are identical then all but one of them are deleted from the equation queue. If any one of the codes is different then unification fails. Following successful processing of ground terms we apply BR operation to reduce the equation queue. It is quite straightforward to extend procedure *CommonPart* to process simulated terms.

5.2. Complexity analysis

We now analyze the complexity of the general unification algorithm developed above. Our analysis proceeds in two stages. First we show that the number of substitutions computed when unifying s and t is at most $O(|s| + |t|)$. Next we show that it is also bounded by $O(k_d k)$ where k_d is the number of distinct variables in s and t together. Hence, the total number of substitutions computed is $O(\min\{(|s|+|t|), k_d k\})$. We also show that counters of all variables can be updated within the above bound. By using the UNION-FIND method in [11], we can merge equation queues in $O(k_d\alpha(k_d))$ time.[5] When k_d is large (say when $O(k_d\alpha(k_d)) \geqslant |s|+|t|$) we can use the technique found in [7] to implement *MergeQ* in $O(|s|+|t|)$ time. Depending on which of the two techniques is used the actual running time of unifying s and t is either $O(\min\{(|s|+|t|), k_{dk}\})$ or $O(|s|+|t|)$ time.

We now establish the bound on number of substitutions computed using a counting technique. With each node we associate an occurrence and substitution counter. The occurrence counter of a node is a count of the number times that node appears in a substitution. This counter is nonzero only when this node is either labeled by an actual variable or represents a virtual variable. The substitution counters are used for amortization of the total number of substitutions computed.

Lemma 14. *The sum of the substitution and occurrence counters of any node labeled by an actual or virtual variable is at most* 2 *and is zero for all other nodes.*

Proof. We use induction on the number of invocations of *CommonPart*.

Base case: Prior to invoking *CommonPart* for the first time, no substitutions have been made and no marks have been placed. The occurrence counter of a node labeled by an actual variable is set to 1. Substitution counters of all nodes are set to zero. So the lemma holds for the base case.

Induction step: Assume that the lemma holds for the first $q-1$ invocations of *CommonPart*. Consider its qth invocation on the terms $\overline{t_1}$ and $\overline{t_2}$. Assume without loss of generality that $\overline{t_1}$ is ahead of $\overline{t_2}$ in the equation queue. Let u be a node in $\overline{t_1}$ labeled by a variable, say x (either virtual or actual) that acquires subterm $\overline{t_3}$ rooted at v in $\overline{t_2}$ as a substitution in this invocation. Bases on the structure of $\overline{t_3}$ we now have three cases to consider.

Case 1: t_3 is a nonground and nonvariable (neither actual nor virtual) subterm (see Fig. 12(a)). By induction hypotheses the sum of occurrence and substitution counters of v is zero. As a result of this substitution we create a new virtual variable at v thereby transforming $\overline{t_2}$ to $\overline{\overline{t_2}}$. So the occurrence counter of v is set to 1. Also observe that the variables $y_1, y_2, \ldots, y_l$ have now been moved to $\overline{t_3}$ which will in turn be placed in E_x. So the occurrence counters of nodes labeled by $y_1, y_2, \ldots, y_l$ will remain unaffected. Finally we account the O(1) cost of computing this substitution by incrementing the substitution counter of u by 1. Observe that $\overline{t_1}$ appears to the left of $\overline{t_2}$ in the equation queue and so it will cease to be an unprocessed term. Therefore the occurrence counter of u is decremented by 1 and hence the sum of the two counters of u is at most 2. The counters of nodes not involved in this substitution remain unaffected and therefore their sum is at most 2.

[5] α is the inverse of Ackermann's function.

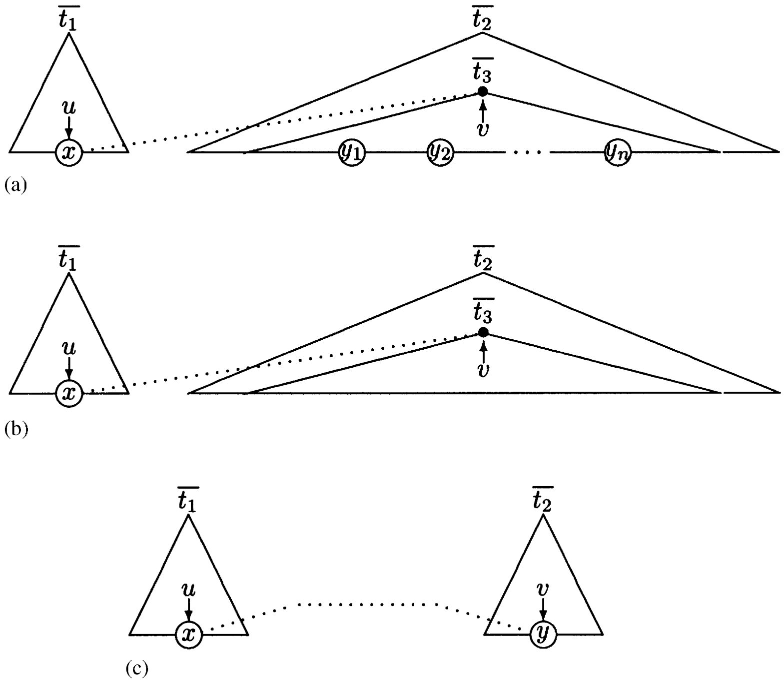


Fig. 12. Situations used in the proof of Lemma 14. (a) Case 1 in the Proof of Lemma 14; (b) case 2 in the Proof of Lemma 14; (c) case 3 in the Proof of Lemma 14.

Case 2: t_3 is a ground subterm (see Fig. 12(b)). No new marks will be placed in this case. The counter of all nodes except that of u remain unaffected. The substitution counter of u is incremented by 1 to absorb the cost of the computed substitution whereas its occurrence counter is decremented by 1. Therefore the sum of the two counters of each node is at most 2.

Case 3: t_3 is a variable (virtual or actual) (see Fig. 12(c)). Changes to the counter are exactly the same as in case 2.

Any substitution computed in this invocation of procedure *CommonPart* falls in one the above three cases and so the lemma holds at end of this invocation. □

From the above result it readily follows that:

Corollary 5. *The number of substitutions computed is at most* $O(|s| + |t|)$.

Lemma 15. *The total number of substitutions computed is at most* $O(k_d k)$.

Proof. Let m be the total number of variables in a equation queue E_x. When processing E_x the same term can appear as one of the input parameters to two successive invocations of procedure *CommonPart*. Suppose t_i is such a such term. Then at the end of *CommonPart*(t_{i-1}, t_i), t_i is modified to $(\overline{t_i})$ by introducing new marks. However, the total number of variables in $(\overline{t_i}) \leqslant$ the total number of variables in t_i. This is because we do not mark ground substitutions. Consequently the number of substitutions computed in *CommonPart*$((\overline{t_i}), t_{i+1})$ cannot be more than that computed in *CommonPart*(t_i, t_{i+1}). This means if there are m variable occurrences among the terms in E_x then in processing E_x we will compute at most $2m$ substitutions.

Now suppose in processing E_x we compute more than $2k$ substitutions. This means that the number of occurrences of variables among the terms in E_x is more than k. As each virtual variable covers at least one actual variable it follows from the proof of Lemma 9 that this unification will fail. So we can terminate processing any equation queue as soon as $2k+1$ substitutions are computed. This means we will compute at most $2k$ substitutions in processing each equation queue in a successful unification. As there are k_d distinct variables the lemma follows. □

Theorem 8. *The total number of substitution computed is* $\mathrm{O}(\min\{k_d k, |s| + |t|\})$.

Proof. Follows from the above lemma and Lemma 14. □

Recall the equation queue E_x is selected for processing when occurrence counter of x becomes 0. From the proof of Lemma 14, note that the occurrence counter of variable is the sum of the occurrence counters of nodes labeled by x (either virtual or actual). Using this it is quite straightforward to implement *UpdateCounters* without increasing the asymptotic complexity of computing the substitutions. Therefore $T_{update} \leqslant T_{cp}$. If *MergeQ* is implemented using the UNION-FIND algorithm [11] then T_{merge} is $\mathrm{O}(k_d k)$. If k_d is very large, i.e., $\mathrm{O}(k_d \alpha(k_d)) > |s| + |t|$ then the technique in [7] can be used to implement *MergeQ* so that T_{merge} is $\mathrm{O}(|s| + |t|)$.

6. Subterm unification

In the previous section we described efficient algorithms to unify s with a subterm t of p. These algorithms are optimized to exploit the number of occurrences of variables in s and t. In this section we show how to integrate these algorithms to do subterm unification. To unify s with subterms of p we first construct two string-matching automata based on the primary strings of s and p. Prior to each unification attempt we will identify the structure of the subterm (i.e., whether it is linear, nonlinear, number of variable occurrences, etc.) and deploy the most efficient algorithm (described in the previous section) appropriate for that structure. In order to identify the number of occurrences of each variable in a subterm, recall that the preorders of the primary terms are stored in arrays. Recall also that with every node v in the array we keep a pointer that points to the variable node closest to v occurring after it. Using this pointer we can visit all the variable nodes in a subterm t in time proportional to the number of variable occurrences in it. In addition, we can also count the number of occurrences of each variable in t within the same time. Thus, we can identify the structure of t in time proportional to the number of variables in it.

6.1. Time complexity

We now analyze the running time complexity of the subterm unification algorithm. Note that at a subterm t of p we apply the unification algorithm most efficient for that term's structure. So the complexity for subterm unification will be the sum of the complexities of the different unification algorithm applied to subterms. We estimate this through a simplified analysis. In this analysis we assume that the unification algorithms applied to the subterms of p are the same. For the analysis to be complete we consider the following cases:

1. s and p are linear. Here we apply the linear–linear unification at each subterm of p.
2. s is nonlinear and p is linear. In this case we apply the linear–nonlinear unification algorithm at each subterm of p.
3. s is linear and every subterm of p is nonlinear. In this case also we apply linear–nonlinear unification algorithm at each subterm of p.
4. s and every subterm of p is nonlinear with at most two occurrences of each variable. In this case we apply the 2-occurrence unification algorithm at each subterms of p.
5. Both s and p are nonlinear and every subterm of p and s have more than two occurrences of each variable. Herein we apply the optimized general nonlinear–nonlinear unification in each attempt.

Observe that from a complexity viewpoint, cases 1 and 5 are the best- and worst-case scenarios and so the actual complexity of subterm unification will lie between these two extremes. But in practice it is unlikely that we reach the bound of 5. This is because in both cases 4 and 5 we have assumed pessimistic scenarios whereas in practice the number of occurrences of a variable in subterms can only decrease as its distance increases from the root of p.

We use the following terminology in the remainder of our analysis. We use n, m, d_p and d_s to denote the size of p, size of s, depth of p and depth of s respectively. We also use k_p, k_t and k_s to denote number of variable occurrences in p, t (a subterm of p) and s respectively.

We now develop the concept of *suffix index* that is used in the description of our complexity results. Let $label(v)$ denote the label of a node v in t. Further let $label(v_i, v_j)$ be q if v_j is the qth child (in the left-to-right order) of v_i. Now suppose $v_1, v_2, \ldots, v_l$ is a sequence of vertices in on the path from v_1 to v_l in t

Definition 3. The labeled path from v_1 to v_l, denoted by $lp(v_1, v_l)$ is the string $label(v_1) \circ label(v_1, v_2) \circ label(v_2) \circ label(v_2, v_3) \ldots label(v_{n-1}) \circ label(v_{n-1}, v_l)$, i.e., $lp(v_1, v_l)$ is a string formed by alternatively concatenating the vertex and edge labels on the path from v_1 to v_l (excluding $label(v_l)$).

Table 2
Table of asymptotic complexities

p \ s	Linear	Nonlinear
Linear	$O(nk_s^* + d_sk_p)$	$O(nk_s^* + d_pk_p)$
Nonlinear	$O(nk_s^* + d_pk_p)$	$\dfrac{\mathit{multiplicity} \leqslant 2}{O(nk_s^* + d_pk_p)}$ $\dfrac{\text{UNION} - \text{FIND } \mathit{MergeQ} \text{ implementation}}{O(\min\{n(m + d_p), k_d(d_pk_p + nk_s)\} + nk_d\alpha(k_d))}$ $\dfrac{\text{Linear } \mathit{MergeQ} \text{ implementation}}{O(n(m + d_p))}$

Let r_s be the root of s and $v_1, v_2, \ldots, v_{k_s}$ be the vertices (in s) that are labeled with variables. Further let $\mathscr{K}_s = \{lp(r_s, v_i) \mid 1 \leqslant i \leqslant k_s\}$. Then,

Definition 4. The suffix number of a string λ in $\mathscr{K}_s$ is the number of strings in $\mathscr{K}_s$ which are suffixes of λ and the *suffix index* of $\mathscr{K}_s$, denoted by k_s^*, is the maximum among the suffix numbers of all strings in $\mathscr{K}_s$. If k_s is 0 then k_s^* is 1.

Table 2 summarizes the complexity results for cases 1–5 discussed above. (In the table, multiplicity is the maximum over the number of occurrences of any variable.) In the following, we establish these results.

6.1.1. Linear–linear subterm unification

Here both s and p are linear. As p is linear every subterm t of p is also linear. Therefore, each unification in this case requires a single invocation of procedure *CommonPart*. Furthermore, the complexity of subterm unification is given by the sum of the substitutions computed over all invocations of *CommonPart*. We now show that:

Lemma 16. *Linear–linear subterm unification computes at most* $O(nk_s^* + d_sk_p)$ *substitutions.*

Proof. We divide the substitutions computed in the subterm unification into two groups. The first group contains substitutions computed for variables in s and the second contains those made for variables in p. We first show that the bound on first group is $O(nk_s^*)$.

Let t be the subterm of p be rooted at v. Now suppose that in the unification of s with t the term rooted at w is computed as a substitution for a variable x in s (see Fig. 13). Based on Theorem 2 we can show that *CommonPart* computes this substitution (even when it terminates with failure) only if

$$lp(r_s, u) = lp(v, w). \tag{1}$$

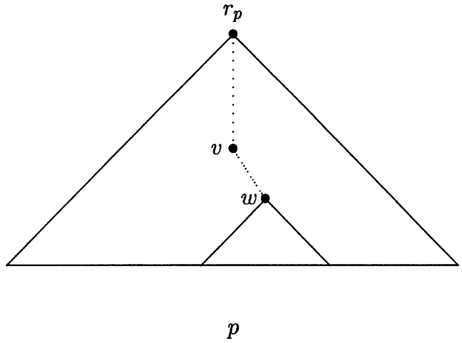


Fig. 13. Bound on number of substitutions computed for variables in s.

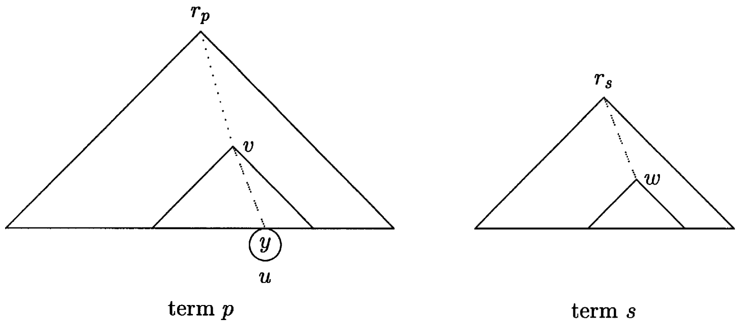


Fig. 14. Bound on number of substitutions computed for variables in p.

In other words, $lp(r_s, u)$ is a suffix of $lp(r_p, w)$. We call subterm rooted at such a w as a *legal* substitution. Note that if the subterm rooted at w is computed as the substitution for some other variable y (in another unification) which is the label of node u_1 in s then $lp(r_s, u_1)$ must also be a suffix of $lp(r_p, w)$. If this is the case either $lp(r_s, u)$ is a suffix of $lp(r_s, u_1)$ or the vice versa. From this we can deduce that the subtree rooted at w can be a legal substitution at most k_s^* times over all unifications. As there are n nodes in p and each can be a legal substitution at most k_s^* times, there can be at most $O(nk_s^*)$ substitutions in the first group.

Now we consider the substitutions in the second group. These are made to variables in p. Let y be a variable in p. Now suppose that during the unification of s and t (rooted at v) the subterm q of s is computed as the substitution for y (see Fig. 14). Let w be the root of q. Then,

$$lp(r_s, w) = lp(v, u). \tag{2}$$

In particular $lp(r_s, w)$ is a suffix of $lp(r_p, u)$. Observe that this condition is satisfied by every substitution computed by *CommonPart* (as all of them are legal substitutions).

We now show that the roots of legal substitutions made to the same variable in p must have distinct depth in s. Assume that this is not true, i.e., assume that two nodes w and $\bar{w}$ of same depth in s are roots of two substitutions computed for y. As w and $\bar{w}$ are distinct nodes having same depth it is clear that $|lp(r_s,w)|=|lp(r_s,\bar{w})|$ and $lp(r_p,w)\neq lp(r_p,\bar{w})$. Therefore both of them cannot be suffixes of $lp(r_p,u)$. Hence only one of can be a legal substitution by (2) above – a contradiction. Therefore each variable in p can acquire at most d_s substitutions. As there are k_p variables in p, the second group contains at most $\mathrm{O}(d_s k_p)$ substitutions.

Therefore the total number of substitutions in both groups together is at most $\mathrm{O}(nk_s^* + d_s k_p)$. □

Theorem 9. *Subterm unification of s with every subterm of p takes at most* $\mathrm{O}(nk_s^* + d_s k_p)$ *time.*

Proof. Follows from the above lemma. □

6.1.2. Linear–nonlinear subterm unification

Herein we have two cases depending on whether s nonlinear or p is nonlinear. However in both cases each unification attempt takes time proportional to the number of substitutions computed in it (see Section 4.1). Therefore, the complexity of subterm unification can again be established by deriving a bound on the total number of substitutions computed. We first derive this bound for the case in which s is nonlinear.

Lemma 17. *The linear–nonlinear subterm unification computes at most* $\mathrm{O}(nk_s^* + d_p k_p)$ *substitutions.*

Proof. As before we divide the substitutions computed into two groups. The first group contains the substitution made to the variables in s and second one contains those made to the variables in the p.

Let t be a subterm of p. As t is linear we may compute multiple substitutions only for variables in s in the invocation *CommonPart*(s,t). Observe that all such substitutions must be subterms of t. Therefore in subsequent common part computations only variables in t can receive substitutions. In other words, the variables in s can receive substitutions only in the first invocation of *CommonPart*. Therefore the substitutions computed for variables in s over all unification attempts must be a legal substitutions. Hence there can be at most $\mathrm{O}(nk_s^*)$ substitutions made for variables in s (see proof of Lemma 16).

Note that the variables in t receive substitutions either during the first invocation of *CommonPart* (i.e., *CommonPart*(s,t)) or during the BR operations applied to solve equation queues (of variables in s). Since t is linear each variable can receive at most one substitution in the invocation *CommonPart*(s,t). Furthermore a variable that receives a substitution in this invocation cannot receive additional substitutions during the BR operations (see Section 4.1). As by Lemma 5, each variable receives at most two substitutions during the BR operations, the number of substitutions computed for variables in t is at most $2 \times k_t$. Using this we now show that the size of the second group is bounded by $2d_p \times k_p$. We do this by induction on the height of p

Base case: When height of p is 1 the result holds trivially.

Induction step: Let us assume that that the result holds for all terms with height less than d_p. Now consider the case when the depth is d_p. Let $p = f(t_1, t_2, \ldots, t_l)$. Further let d_i and k_i denote the depth and number of variables in t_i $(1 \leqslant i \leqslant l)$. Now the subterm unification of s and p is performed by first unifying s and p (at p's root) and then by applying l subterm unifications to unify s with subterms of every t_i's. Observe that the height of each t_i is smaller than d_p. Therefore by the induction hypothesis, in the subterm unification of s and t_i there are at most $2d_ik_i$ substitutions in the second group. By Lemma 5 while unifying s and p we compute at most 2 substitutions for each variable in p. Let $R(q,s)$ and $T(q,s)$ denote the total number of substitutions (in the second group) computed in the unification and subterm unification of s and q respectively. Then,

$$\begin{aligned} T(p,s) &= R(p,s) + \sum_{i=1}^{i=l} T(t_i,s) \\ &\leqslant 2 * k_p + \sum_{i=1}^{i=l} 2 * d_i * k_i. \end{aligned}$$

Let $\bar{d}$ be the largest among d_i's. Since t_i's are nonoverlapping subterms of s, $d_p = \bar{d} + 1$ and $k_p = \sum_{i=1}^{i=l} k_i$. Using these we get

$$\begin{aligned} T(p,s) &\leqslant 2 * k_p + \sum_{i=1}^{i=l} 2 * \bar{d} * k_i \\ &\leqslant 2 * k_p + 2 * (d_p - 1) * \sum_{i=1}^{i=l} k_i \\ &\leqslant 2 * k_p + 2 * (d_p - 1) * k_p = 2 * d_p * k_p. \end{aligned}$$

Hence the lemma. □

Theorem 10. *Subterm unification of p with subterms of s takes* $O(nk_s^* + d_pk_p)$.

Proof. Follows from the above lemma, Corollary 2 and Lemma 8. □

The complexity of subterm unification for the second case, namely for nonlinear p and linear s, is given by the following theorem.

Theorem 11. *If s is linear and p is nonlinear then linear–nonlinear subterm unification requires at most* $O(nk_s + d_sk_p)$ *time.*

Proof. Similar to the proof of Lemma 17. □

6.1.3. Nonlinear–nonlinear subterm unification

We begin with the simple case wherein we assume that each unification involves terms that contain at most two occurrence of each variable. As each unification attempt can be solved by the 2-occurrence unification algorithm we have:

Theorem 12. *Subterm unification with* 2-*occurrence unification requires at most* $O(nk_s + d_pk_p)$ *time.*

Proof. By Theorem 6 each unification attempt takes time proportional to the number of variable occurrences in the terms being unified. We can divide the substitutions into two groups as done before and obtain the bounds on them. Since there are k_s variables in s and n unification attempts, the number substitutions computed for variables in s is nk_s. By using induction on height of p (as done in Lemma 17) we can establish that the number of substitution made to variables in p is bounded by d_pk_p. Hence the result. $\square$

Now we consider the most pessimistic scenario in which each unification attempt requires the general unification algorithm developed in Section 5. In this case *MergeQ* can be implemented in two different ways. Suppose we use UNION-FIND method then unification of s and (subterm) t requires $O(\min\{|s| + |t|, k_1(k_s + k_t)\} + k_1\alpha(k_1))$ where k_1 is the number of distinct variables in s and t. On the other hand if we use the method outlined in [7] then each unification requires $O(|s| + |t|)$ time. Although it is possible to mix the two implementations in a single subterm unification for our analysis we will consider them separately. Suppose we implement *MergeQ* using the method in [7]. By using induction on p (as done in the proof of Lemma 17) we can show that subterm unification requires at most $O((m+d_p)n)$ time. Now suppose we use the UNION-FIND approach to implement *MergeQ*. Further assume that the number of distinct variables in s and any subterm t is the same as that in s and p (i.e., k_d).

Lemma 18. *With UNION-FIND implementation of MergeQ, subterm unification requires* $O(\min\{(m + d_p)n, k_d(d_pk_p + nk_s)\} + nk_d\alpha(k_d))$ *time.*

Proof. Assuming that in each unification $\min\{|t| + m, k_d(k_t + k_s)\}$ is $|t| + m$ and summing up this quantity over all unifications we get $(m + d_p)n$. Similarly assuming that $\min\{|t|+m, k_d(k_t+k_s)\}$ is $k_d(k_t+k_s)$ we get $k_d(d_pk_p+nk_s)$. As we assume that k_d remains the same in all n unification attempts, the complexity of subterm unification is at most $O(\min\{(n + d_p)m, k_d(d_pk_p + nk_s)\} + nk_d\alpha(k_d))$. $\square$

7. Conclusions

We presented an algorithm for efficient subterm unification. The basic idea underlying the algorithm is to exploit the commonality among subterms. Our algorithm uses a suite of techniques that are deployed in such a way that in most cases it performs much better than applying the linear time unification algorithms in [7, 8] at each subterm. Furthermore our algorithm is guaranteed to perform no worse even in the most pessimistic scenario. The techniques used in our algorithm are also potentially useful in problems where one term is repeatedly unified with a set of terms.

References

[1] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, CACM 18 (6) (1975) 333–340.

[2] P.J. Downey, R. Sethi, R.E. Tarjan, Variations on the common subexpression problem, J. Appl. Comput. Math. 24 (4) (1980) 758–771.

[3] C.H. Hoffmann, M.J. O'Donnell, Pattern matching in trees, J. Appl. Comput. Math. 29 (1) (1982) 68–95.

[4] G. Huet, J.-J. Levy, Computations in orthogonal rewriting systems, in: J.-L. Lassez, G. Plotkin (Eds.), Essays in Computational Logic (in honor of Alan Robinson), MIT Press, Cambridge, MA, 1991.

[5] D.E. Knuth, P. Bendix, Simple word problems in universal algebras, in: J. Leech (Ed.), Computational Problems in Abstract Algebra, Pergammon Press, Oxford, 1970, pp. 263–297.

[6] A. Martelli, U. Montanari, An efficient unification algorithm, ACM TOPLAS 4 (2) (1982) 258–282.

[7] A. Martelli, U. Montanari, Unification in linear time and space: a structured presentation, internal report #B76-16, Instituto di Elaborazione della Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy, July 1976.

[8] M.S. Paterson, M.N. Wegman, Linear unification, J. Comput. System Sci. 16 (2) (1978) 158–167.

[9] R. Ramesh, I.V. Ramakrishnan, Nonlinear pattern matching in trees, J. Appl. Comput. Math. 39 (2) (1992) 295–316.

[10] R. Ramesh, I.V. Ramakrishnan, Automata-driven indexing of prolog clauses, Technical Report UTDCS2092, Department of Computer Science, University of Texas at Dallas, (To appear in J. Logic programm.), Also available through anonymous ftp as ftp.utdallas.edu:/pub/cs/ramesh/jlp.dvi.

[11] R.E. Tarjan, On the efficiency of a good but not linear set merging algorithm, J. Appl. Comput. Math. 22 (2) (1975) 215–225.