Unification Factoring for Efficient Execution of Logic Programs*

S. Dawson C.R. Ramakrishnan I.V. Ramakrishnan K. Sagonas
S. Skiena T. Swift D.S. Warren

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400

{sdawson, cram, ram, kostis, skiena, tswift, warren}@cs.sunysb.edu

Abstract

The efficiency of resolution-based logic programming languages, such as Prolog, depends critically on selecting and executing sets of applicable clause heads to resolve against subgoals. Traditional approaches to this problem have focused on using indexing to determine the smallest possible applicable set. Despite their usefulness, these approaches ignore the non-determinism inherent in many programming languages to the extent that they do not attempt to optimize execution *after* the applicable set has been determined.

Unification factoring seeks to rectify this omission by regarding the indexing and unification phases of clause resolution as a single process. This paper formalizes that process through the construction of *factoring automata*. A polynomial-time algorithm is given for constructing optimal factoring automata which preserve the clause selection strategy of Prolog. More generally, when the clause selection strategy is not fixed, constructing such an optimal automaton is shown to be NP-complete, solving an open trie minimization problem.

Unification factoring is implemented through a source code transformation that preserves the full semantics of Prolog. This transformation is specified in the paper, and using it, several well-known programs show performance improvements of up to 100% across three different systems. A prototype of unification factoring is available by anonymous ftp.

1 Introduction

In logic programming languages, such as Prolog, a predicate is defined by a sequence of Horn clauses. When resolving a goal, a clause becomes applicable if its head unifies with the goal, and each applicable clause is invoked in textual order. Unification of a clause head with a goal involves two basic operations: elementary match operations and computing substitutions for variables in the two terms. When there are common parts among the clause heads, it should be possible to share the basic operations corresponding to these common parts and do them without redundancy. Developing such techniques is a problem of considerable importance for efficient evaluation of logic programs.

Traditionally this optimization is viewed as a two-phase process. The first phase, known as the *indexing* phase, examines non-variable parts of the goal and clause heads to compute a *match set* which is a superset of all unifiable clauses. While indexing essentially does match operations, the substitutions are computed in the second, or *unification*, phase when the goal is unified with the clauses in the match set. For example, indexing yields the three clauses $\{p(a,b,c), p(a,b,d), p(a,c,c)\}$ for the call $p(a,X,Y)$ on the predicate in Figure 1a. In the unification phase each of the three clauses is unified in textual order. In this phase six substitutions are computed – three for each of the two variables. In addition three match operations are performed requiring rescanning of terms already seen during indexing. But observe that it suffices to compute only two substitutions for X and three for Y. Furthermore repeating the match operation is unnecessary. Thus the efficiency of unification of clause heads with the goal can be considerably enhanced by sharing the unification operations not only in the indexing but also in the unification phase.

Although techniques for sharing tests needed for computing match sets have been extensively researched (such as indexing techniques for logic programming, *e.g.,* see [2, 5, 6, 8, 9, 14]; pattern matching for functional and term rewriting systems *e.g.,* see [11]; and decision trees for concurrent logic languages *e.g.,* [7, 12]), optimizing the sharing of unification operations has not been explored. Extant indexing techniques in logic programming, on completing indexing, unify each clause head in the match set separately with the goal. That is, execution after indexing is not optimized. Even rescanning parts already seen during indexing is seldom avoided (see [2, 5]) since in general this requires either large indexing structures (*e.g.,* the switching tree of [6]) or elaborate information to be maintained and manipulated at run time. In any case since each clause head is unified separately with the goal, the other problem of sharing substitutions still remains.

Rather than viewing head unification as two separate and independent stages we regard it as a single process in which both the indexing and unification phases are blended. We present a technique, called *unification factoring*, to unify clause heads efficiently with any *arbitrary* goal. In contrast to matching trees (*e.g.,* [6, 14]), the technique presented here does not rely on mode information. Unification factoring transforms the source program into another in which

*This work was supported in part by NSF grants CCR-9102159, CCR-9102989, CCR-9404921, CDA-9303181, INT-9314412, and ONR grant 400X116YIP01.

the basic unification operations common to clause heads and the goal are *factored out*; *i.e.,* they can all be shared. For instance, the program in Figure 1a can be transformed into the program in Figure 1b by unification factoring. Now the call $p(a, X, Y)$ on this transformed predicate will result in only one match operation and compute two substitutions for X; the three substitutions computed for Y remain unchanged. Observe that in the transformed program, the match operation is shared and only the needed substitutions are computed for X.

```
p(a,b,c).
p(a,b,d).
p(a,c,c).
p(b,a,c).
```

(a)

```
p(a,X,Y) :- p1(X,Y).
p(b,a,c).
p1(b,X) :- p2(X).
p1(c,X) :- p3(X).
p2(c).
p2(d).
p3(c).
```

(b)

Figure 1: Original predicate (a) and transformed version (b)

Unification factoring is a uniform technique that can enhance the overall efficiency of unifying clause heads with a goal by optimizing not just the matching operations (as is done by indexing techniques) but also other operations (such as computing substitutions). In other words, it can handle non-deterministic execution more efficiently. Moreover, since there is no division into two separate phases, the difficult interface problem of eliminating rescans of terms no longer exists. In general there are several different ways to factor out unification operations in a program, yielding transformed programs with differing performance. The interesting problem then is the design of *optimal* unification factoring, *i.e.,* one that results in a program that has the *best* performance. We propose a solution to this problem. In contrast to existing indexing techniques, which require compiler and/or engine modifications, unification factoring is implemented as a source-to-source transformation needing no engine changes. We develop the technique of unification factoring in two parts. In the first part we construct a *factoring automaton* that models the process of unifying a goal with a set of clause heads as is done in the WAM (see Section 2). Common unification operations are factored out by this automaton. The second part constitutes the algorithm for transforming this automaton into Prolog code (see Section 4).

Summary of Results

1. We describe an algorithm for constructing (at compile time) an optimal factoring automaton that faithfully models Prolog's textual order-preserving clause selection strategy. We exploit this strategy for constructing an optimal automaton in polynomial time using dynamic programming (see Section 3).
2. We show that, on relaxing the order-preserving strategy, construction of an optimal automaton becomes NP-complete (see Section 3). This result solves a trie minimization problem left open by Comer and Sethi [4].
3. We provide experimental evidence that our transformation can consistently improve speeds of Prolog programs by factors up to 2 to 3 on widely available Prolog systems, namely, Quintus, SICStus, and XSB (see Section 5). Our results also indicate that, although the transformation can in principle increase code size by at most a constant factor, in practice this increase is never more than 10%, and in fact, code space decreases in some cases.

2 Unification Factoring

The factoring automaton decomposes the unification process into a sequence of elementary unification operations that model instructions in the WAM. It is structured as a tree, with the root as the start state, and the edges, denoting transitions, representing elementary unification operations. Each transition is associated with the cost of performing the corresponding operation. Every leaf state represents a clause, and the transitions on the path from the root to a leaf represent the set of elementary operations needed to unify the head of that clause with a goal. The total cost of all these transitions is the cost of this unification. Common edges in the root-to-leaf paths of two leaves represent common operations that are needed to unify the goal with those two clauses. Sharing the operations associated with common edges thus amounts to factoring the process of unifying the goal with the clause heads. Note that, since the transitions represent unification operations, all possible transitions out of a state are attempted; *i.e.*, the automaton is non-deterministic. In the following, we formalize the notion of factoring automaton. Our formalization is inspired by work on pattern matching tries in [1]. In the next section we describe the construction of optimal automata.

The Factoring Automaton

We assume the standard definitions of term, and the notions of substitution and subsumption of terms. A *position* in a term is either the empty string Λ that reaches the root of the term, or $\pi.i$, where π is a position and i is an integer, that reaches the i^{th} child of the term reached by π. By $t|_\pi$ we denote the symbol at position π in t. For example, $p(a, f(X))|_{2.1} = X$. We denote the set of all positions by Π. Terms are built from a finite set of function symbols $\mathcal{F}$ and a countable set of variables $\mathcal{V} \cup \hat{\mathcal{V}}$, where $\hat{\mathcal{V}}$ is a set of position variables. The variables in the set $\hat{\mathcal{V}}$ are of the form X_π, where π is a position, and are used simply as a notational convenience to mark certain positions of interest in a term. The symbol t (possibly subscripted) denotes terms; $\alpha, \alpha', \ldots$ denote elements of the set $\mathcal{F} \cup \mathcal{V}$; $\gamma, \gamma', \ldots$ denote elements of the set $\mathcal{F} \cup \mathcal{V} \cup \Pi$; and f, g, h denote function symbols. The arity of a symbol α is denoted by $arity(\alpha)$; note that the arity of variable symbols is 0. Simultaneous substitution of a term t' at a set of positions P in term t is denoted $t[P \leftarrow t']$. For example, $p(X_1, f(X_{2.1}), X_3)[\{2.1, 3\} \leftarrow b] = p(X_1, f(b), b)$.

A factoring automaton performs unification as a series of elementary unification operations. At each stage in the computation we need to capture the operations that have been performed, as well as those that remain to be done. We use the notion of *skeleton*, which is a term over $\mathcal{F} \cup \mathcal{V} \cup \hat{\mathcal{V}}$, to denote this partial computation. Elements of $\mathcal{F} \cup \mathcal{V}$ in a skeleton represent unification operations that have been performed. Position variables denote portions of the goal where the remaining operations will be performed. Given a skeleton its *fringe* defines the positions to be explored for unification to progress. Formally,

Definition 2.1 (Skeleton and Fringe) A *skeleton* is a term over $\mathcal{F} \cup \mathcal{V} \cup \hat{\mathcal{V}}$. The *fringe* of a skeleton S, denoted *fringe*(S), is the set of all positions π such that $S|_\pi = X_\pi$ and $X_\pi \in \hat{\mathcal{V}}$.

For example, the skeleton $q(X_1, g(X_{2.1}, X_{2.3}, X_{2.3}))$ for the goal $q(f(U), W)$ captures the fact that the substitution for W has been partially computed (to be $g(X_{2.1}, X_{2.3}, X_{2.3})$), and that the first argument of the goal has not yet been explored. The fringe of this skeleton is $\{1, 2.1, 2.3\}$.

Each state in the automaton represents an intermediate stage in the unification process. With each state is associated a skeleton and a subset of clauses, called the *compatible set* of the state. The clause heads in the compatible set share each unification operation done on the path from the root to that state. Hence, each clause head in the compatible set is subsumed by the skeleton of that state. Recall that the fringe of a skeleton represents the positions in the partially unified goal that remain to be explored. A state then specifies one such position, and each outgoing transition represents a unification operation involving that position. We label the transition *unify*(π, γ), where γ is either a function symbol or variable in the clause head, or another position in the (partially unified) goal. (Positions in the label are prefixed with "\$" to avoid confusion with integers.)

For example, in Figure 2a the label on the transition from s_1 to s_2 specifies unifying position 1 of the goal with a. The compatible set for state s_2 is $\{p(a,b,c), p(a,b,d), p(a,c,c)\}$, and clause heads in that set share the operation *unify*$(\$1, a)$. In Figure 5b the label on the transition from s_1 to s_2 specifies unifying positions 1 and 2 in the goal, while the label on the transition from s_2 to s_5 specifies unifying position 2 in the goal with variable X (in the head of clause 1).

A transition from a state indicates progress in the unification process. For a transition labeled *unify*(π, γ), the skeleton of the destination state is obtained by extending the skeleton S of the current state using the extension operation $extend(S, \pi, \gamma)$ defined below. Intuitively, S is extended by replacing all occurrences of X_π in S by the term corresponding to γ. If γ is a function symbol, this term has γ as root and position variables representing new fringe positions as its children. If γ is a position, this term is the position variable X_γ. Otherwise, this term is the variable γ itself.

Definition 2.2 (Skeleton extension) The *extension* of skeleton S at fringe position π by γ, denoted $extend(S, \pi, \gamma)$, is the skeleton S' such that

$$\begin{aligned} S' &= S[P \leftarrow t] \\ \textit{where}\quad P &= \{\pi' \mid S|_{\pi'} = X_\pi\} \\ \textit{and}\quad t &= \begin{cases} \gamma(X_{\pi.1}, \ldots, X_{\pi.arity(\gamma)}) & (\gamma \in \mathcal{F}) \\ X_\gamma & (\gamma \in \Pi) \\ \gamma & (\gamma \in \mathcal{V}) \end{cases} \end{aligned}$$

For example, in the skeleton $q(X_1, g(X_{2.1}, X_{2.3}, X_{2.3}))$, the operation *unify*$(\$1, f/1)$ results in extension of the skeleton to $q(f(X_{1.1}), g(X_{2.1}, X_{2.3}, X_{2.3}))$. This skeleton is further extended as a result of the operation *unify*$(\$1.1, \$2.1)$ to $q(f(X_{2.1}), g(X_{2.1}, X_{2.3}, X_{2.3}))$.

We now formally define the factoring automaton as follows:

Definition 2.3 (Factoring Automaton) A *factoring automaton* for a set of clauses C and a skeleton S is an ordered tree whose edges are labeled with *unify*(π, γ), and with each node s (a state) is associated a skeleton S_s, a position $\pi_s \in$ *fringe*(S_s) (if s is not a leaf), and a non-empty *compatible* set $C_s \subseteq C$ of clause heads such that:

1. Every clause head in C_s is subsumed by S_s,
2. the root state has S as the skeleton and C as the compatible set,
3. for each edge (s, d) with label *unify*(π_s, γ), $S_d = extend(S_s, \pi_s, \gamma)$, and
4. the collection of sets $\{C_d \mid (s, d) \text{ is an edge}\}$ is a partition of C_s.

The partitioning of the compatible set C_s at a fringe position π_s in the above definition ensures that transitions specify unification operations involving π_s in the goal, and either: 1) a function symbol or variable appearing at π_s *in at least one* of the clause heads in C_s; or 2) another (fringe) position in the goal. Each set in the partition is a compatible set of one of the next states of S_s, and all the clause heads in it share the unification operation specified by the corresponding transition.

Construction Using the programs in Figures 1a and 5a and the corresponding automata in Figures 2a and 5b for illustration, we informally describe the construction of a factoring automaton. For a predicate p/n an automaton is built incrementally starting with the skeleton $p(X_1, \ldots, X_n)$ for the root state s_1 and the set of clauses defining p/n as its compatible set. From a given state s we "expand" the automaton as follows. We first choose a position π_s from the fringe of its skeleton S_s. We then partition the compatible set C_s into sets $C_{d_1}, \ldots, C_{d_k}$ such that in each C_{d_i}, all clause heads specify the same unification operation, *unify*(π_s, γ_i), at π_s. For example, at state s_2 in Figure 2a, the compatible set $\{p(a,b,c), p(a,b,d), p(a,c,c)\}$ is partitioned into $\{p(a,b,c), p(a,b,d)\}$, which share the unification operation *unify*$(\$2, b)$, and $\{p(a,c,c)\}$, which has operation *unify*$(\$2, c)$. We create new states $d_1, \ldots, d_k$ such that for each state d_i, C_{d_i} is its compatible set, $S_{d_i} = extend(S_s, \pi_s, \gamma_i)$ is its skeleton, and the edge (s, d_i) is the transition into d_i, labeled with *unify*(π_s, γ_i). The process of expanding the automaton is repeated on all states that have non-empty fringes.

To partition the clauses, we identify the set of possible unification operations for each clause head at a given fringe position π. For a linear clause head t, the only possible unification operation involves the symbol at position π in the clause head, *i.e.*, *unify*$(\pi, t|_\pi)$. For a non-linear clause head, there is an additional possible operation for each fringe position in the head having a term identical to that at position π. That is, for each fringe position $\pi' \neq \pi$, such that the subterms rooted at π and π' are identical, *unify*(π, π') is also a possible operation. Two clause heads may then be included in the same partition iff their respective sets of possible unification operations contain an identical[1] operation.

[1] Variable symbols in distinct clauses are assumed to be distinct.

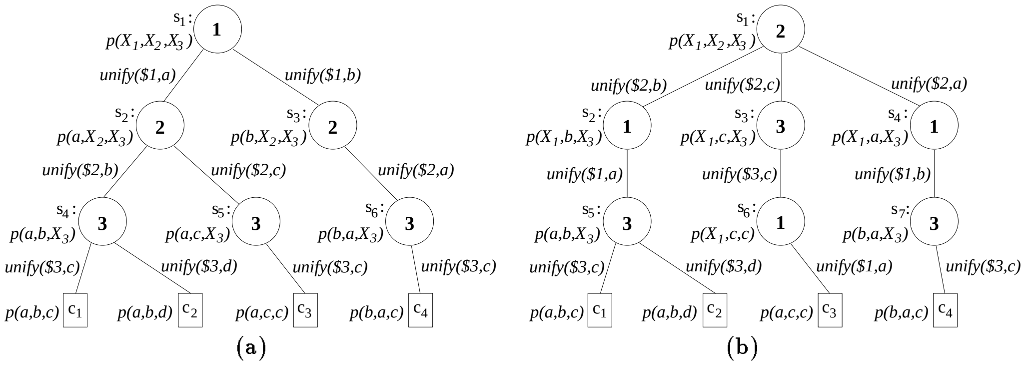


Figure 2: Factoring automata for program in Figure 1a

Operation Unification of a goal with the clause heads begins at the root of the automaton. From each state, a transition is made to the next state by performing the specified unification operation on the partially unified goal. If more than one transition is possible, the first such transition is made, and the remaining transitions are marked as pending. When a leaf is reached, the body of the corresponding clause is invoked. Whenever failure occurs, either within the automaton (due to failure of a unification operation), or during execution of a clause body, the automaton backtracks to the nearest state with pending transitions. The process then continues with the next pending transition. The automaton fails on backtracking when there are no states with pending transitions.

For example, in Figure 2a for the goal $p(a, b, X)$, only the transition labeled *unify*($\$1, a$) is possible from state s_1. Similarly, from state s_2 only the transition labeled *unify*($\$2, b$) is possible. At state s_4, since both outgoing transitions from this state are possible, the transition labeled *unify*($\$3, d$) is marked as pending, and the transition labeled *unify*($\$3, c$) is made, thus unifying X with c and invoking the body of clause 1. Upon backtracking (to state s_4), the transition labeled *unify*($\$3, d$) is taken, unifying X with d and invoking the body of clause 2. Observe that unifications of the goal with the heads of clauses 1 and 2 share the operations of unifying argument 1 with a and argument 2 with b. The above informal description of the automaton's operation can be formalized and its soundness and completeness can be readily established; these are routine and omitted.

Note that a factoring automaton as defined above does not adhere to the order-preserving clause selection strategy of Prolog. For example, on query $p(X, Y)$ the answers computed by the program in Figure 3a appear in the order $\langle p(a, b), p(b, c), p(a, d)\rangle$, while the factoring automaton in Figure 3b for the same program computes the answers in the order $\langle p(a, b), p(a, d), p(b, c)\rangle$. To preserve clause order, each state in the automaton must consider its compatible clauses as a *sequence* and partition this sequence into a collection of subsequences. For this we introduce the following concept of *sequential factoring automaton* (SFA).

Definition 2.4 (Sequential Factoring Automaton) A *sequential factoring automaton* for a sequence $\langle c_1, c_2, \ldots, c_n\rangle$ of clauses is a factoring automaton A such that, in a left-to-right preorder traversal of A, leaf i is visited before leaf $i + 1$, for $1 \leq i < n$.

Figure 3c shows an SFA for the program in Figure 3a. We refer to factoring automata that are not sequential as *non-sequential factoring automata* (NSFA).

3 Optimal Automata

The cost of unifying a goal with the clause heads depends on both the goal and the automaton with which the unifications are performed. For instance, for the goal $p(a, X, c)$, the automaton given in Figure 2a performs three matches and two bindings, whereas the automaton in Figure 2b performs five matches and three bindings. The unification of the goal $p(X, b, c)$, on the other hand, requires three matches and two bindings in the automaton of Figure 2a, whereas the automaton in Figure 2b performs only two matches and one binding. Thus, the relative costs of automata vary with the goal. We assume no knowledge of the goal at compile time, and hence, we choose the *worst case* performance of an automaton as the measure of its cost. An optimal automaton, hence, is one with the lowest worst-case cost[2]. We exploit the order-preserving clause selection strategy of SFAs for constructing an optimal SFA in polynomial time. On the other hand, when clause order is not preserved, as in an NSFA, we show that constructing an optimal automaton is NP-complete.

Optimal SFA

The following three readily established properties of an optimal SFA are used in its construction.

Property 1 Each sub-automaton is optimal for the clauses in the compatible set and skeleton associated with the root of the sub-automaton.

Property 2 Any unification that *can* be shared *will* be shared. That is, in an optimal SFA, no two adjacent transitions from any given state have the same label.

Property 3 Transitions from a state partition its compatible set of clauses into *subsequences*.

To construct an optimal automaton for a sequence of clauses C and initial skeleton S, we consider, for each position in the fringe of S, the least cost automaton that can be constructed by first inspecting that position. From Properties 2

[2] Whereas an optimal factoring automaton minimizes the total number of unification operations over all clause heads, an optimal decision tree minimizes the number of tests needed to identify any one clause.

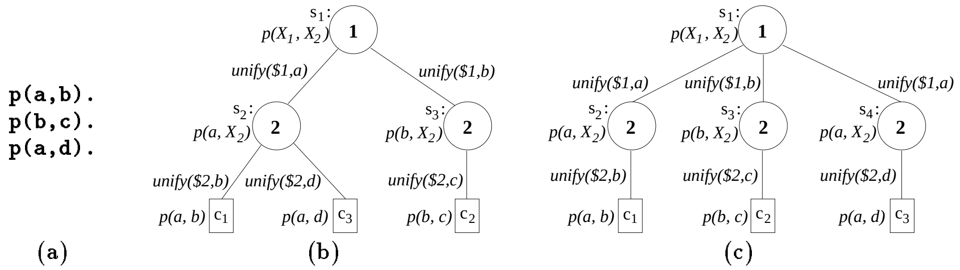


Figure 3: Predicate (a) and non-sequential (b) and sequential (c) automata

and 3 it follows that the transitions out of a state and their order are uniquely determined by the position chosen. Since these transitions represent the partition of the compatible set of clauses, to construct an optimal automaton for a given start position, we first compute this partition. We then construct optimal automata for each sequence of clauses in the partition, and the corresponding extended skeletons. From Property 1 it follows that these optimal sub-automata can be combined to produce an optimal automaton for the selected start position. The lowest cost automaton among the automata constructed for each position is an optimal SFA for C and S. The recursive construction sketched above lends itself to a dynamic programming solution. Below we formalize the construction by first considering linear clause heads. Construction in the presence of non-linear clause heads is discussed later.

Linear clause heads We use a function *part* to partition the clause sequence C into the minimum number of subsequences that share unification operations at this position.

Definition 3.1 (Partition) Given a sequence $\langle t_1, \ldots, t_n \rangle$ of clause heads corresponding to the sequence of clauses C, a pair of integers (i, i'), $1 \le i \le i' \le n$, and a position π, the *partition of C by π*, denoted $part(i, i', \pi)$, is the set of triples (α, j, j'), $i \le j \le j' \le i'$, such that j and j' are the end points of a maximal subsequence of clause heads in $\langle t_i, \ldots, t_{i'} \rangle$ having symbol α at position π. That is, $(\alpha, j, j') \in part(i, i', \pi)$ iff

1. for all $j \le k \le j'$, $t_k|_\pi = \alpha$,
2. either $j = i$ or $t_{j-1}|_\pi \ne \alpha$, and
3. either $j' = i'$ or $t_{j'}|_\pi \ne \alpha$.

Each triple (α, j, j') computed by *part* represents a transition associated with the unification operation involving π and α. For example, the partition of the sequence of clause heads $\langle p(a,b), p(b,c), p(a,d) \rangle$ at position 1 is the set of subsequences $\{\langle p(a,b) \rangle, \langle p(b,c) \rangle, \langle p(a,d) \rangle\}$. The skeleton of the next state resulting from the transition is $extend(S, \pi, \alpha)$. The compatible clauses of this state form the subsequence $\langle C_j, \ldots, C_{j'} \rangle$.

For each subsequence in a partition there are one or more fringe positions in the skeleton with a symbol common to all clause heads in the subsequence. From Property 2 it follows that the unification operations at these positions will be shared by all clause heads in the subsequence. We use the function *common* to identify such operations and extend the skeleton to record their effect:

Definition 3.2 (Common) Given a skeleton S, a sequence $\langle t_1, \ldots, t_n \rangle$ of clause heads, and a pair of integers (i, i'), $1 \le i \le i' \le n$, $common(S, i, i')$ is a pair (E, S'), where E represents the set of unification operations common to $\{t_i, \ldots, t_{i'}\}$, and S' is the extended skeleton:

$$
\begin{aligned}
&common(S, i, i') = \\
&\qquad (E \cup \{(\pi, \alpha)\}, S'), \quad \text{if } \exists \pi \in fringe(S) \text{ such that} \\
&\qquad\qquad\qquad\qquad t_j|_\pi = \alpha,\ i \le j \le i', \\
&\qquad\qquad\qquad\quad \text{where } (E, S') = \\
&\qquad\qquad\qquad\qquad common(extend(S, \pi, \alpha), i, i') \\
&\qquad (\{\,\}, S), \text{ otherwise}
\end{aligned}
$$

For example, given skeleton $S = p(X_1, X_2, X_3)$ and the sequence $\langle p(a,b,c), p(a,b,d), p(a,b,e) \rangle$, $common(S, 1, 3) = (\{(1,a), (2,b)\}, p(a, b, X_3))$.

The worst case cost of an SFA is when all transitions are taken. Assuming that all elementary unification operations have unit cost, the cost of an optimal SFA for clause sequence $\langle C_i, \ldots, C_{i'} \rangle$ and skeleton S is expressed by Equation 1 in Figure 4. Note that $|E|$ is the number of common unification operations for a subsequence in a partition. Also note that the recurrence assumes that subsequence $\langle t_i, \ldots, t_{i'} \rangle$ has no common part with respect to skeleton S. Thus, the cost of an optimal automaton for a predicate p/m consisting of n clauses is given by $|E| + simple\_cost(1, n, S)$, where $(E, S) = common(p(X_1, \ldots, X_m), 1, n)$.

Using Equation 1, it is straightforward to construct an optimal SFA based on dynamic programming, where an optimal automaton for each subsequence of clauses is recorded in a table. A complete example of this construction is given in the appendix. Note that, since $\langle t_i, \ldots, t_{i'} \rangle$ has no common part with respect to the skeleton, a skeleton S is uniquely determined, given i and i'. Hence, the cost recurrence has only two independent parameters, i and i', and the table used in the dynamic programming algorithm will be indexed by i and i'.

Given n clauses with at most m symbols in each clause head, the number of possible subsequences is $O(n^2)$, and hence, the number of table entries is $O(n^2)$. The number of partitioning positions considered for each entry is $O(m)$. For each position, *part* requires $O(n)$ time. Identification of common operations for all subsequences in a partition can be accomplished in $O(m)$ time via a precomputed matrix (that requires $O(mn)$ time and space). Thus, the time required to compute one entry in the table is $O(m(n+m))$, and the overall time needed to compute an optimal automaton is $O(n^2 m(n+m))$. Therefore,

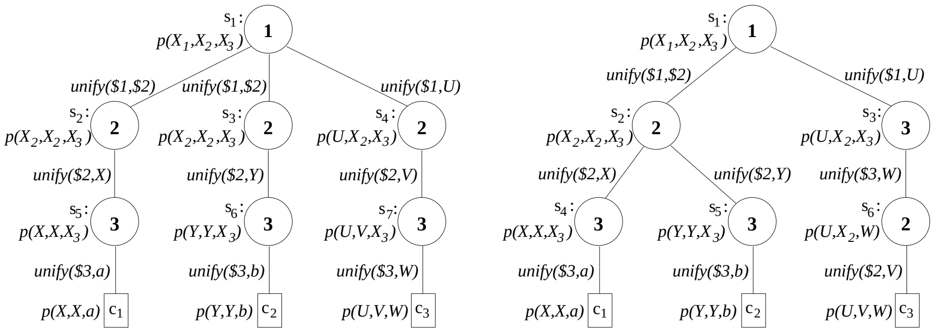
Theorem 3.3 *An optimal SFA can be constructed in polynomial time.*

$$simple\_cost(i,i',S) = \min_{\pi \in fringe(S)} \Big(\sum_{\substack{(\alpha,j,j') \in \\ part(i,i',\pi)}} (simple\_cost(j,j',S') + |E|, \text{ where } (E,S') = common(S,j,j')) \Big) \quad (1)$$

$$refined\_cost(i,i',S) = \min_{\pi \in fringe(S)} \Big(cost_{choice}(\pi) + \sum_{\substack{(\alpha,j,j') \in \\ part(i,i',\pi)}} (refined\_cost(j,j',S') + \sum_{(\pi',\alpha') \in E} cost_{unify}(\pi',\alpha')) \Big) \quad (2)$$
$$\text{where } (E,S') = common(S,j,j')$$

Figure 4: Simple (1) and refined (2) recurrences for cost of optimal SFA

```
p(X,X,a).
p(Y,Y,b).
p(U,V,W).
```




```
p(X,X,Y) :-
    p1(X,Y).
p(U,V,W).
p1(X,a).
p1(X,b).
```

(a) (b) (c) (d)

Figure 5: Non-linear predicate (a), automata (b) and (c), and transformation (d)

In Equation 1 for the cost of an optimal SFA, it is assumed that all unification operations have unit cost, and that recording pending transitions has zero cost. The recurrence can be modified as follows to account for these costs. Let $cost_{unify}(\pi, \alpha)$ be the cost of the unification operation involving position π and α and $cost_{choice}(\pi)$ represent the cost of choosing a transition when multiple transitions are possible at position π. Note that the cost of recording pending transitions can be modeled using $cost_{choice}(\pi)$. Accounting for these costs, an optimal SFA can be computed using Equation 2 (Fig. 4).

Nonlinearity With suitable modification of functions *part* and *common*, Equations 1 and 2 can be used even in the presence of nonlinear clause heads. Consider the program in Figure 5a. The dynamic programming algorithm for linear clause heads yields the automaton in Figure 5b. Observe that this automaton does not share the operation that unifies arguments 1 and 2 that is common to clauses 1 and 2. These operations can be shared by considering relationships among different positions within a clause head. For this we view each clause head as a set of equations. For example, the three clause heads in Figure 5a correspond to the following three sets of equations:

$$\begin{array}{lll} \texttt{p(X,X,a).} & \rightarrow & \{X_1 = X, X_2 = X, X_3 = a, X_1 = X_2\} \\ \texttt{p(Y,Y,b).} & \rightarrow & \{X_1 = Y, X_2 = Y, X_3 = b, X_1 = X_2\} \\ \texttt{p(U,V,W).} & \rightarrow & \{X_1 = U, X_2 = V, X_3 = W\}. \end{array}$$

The unification operations involving position 1 are $X_1 = X$ and $X_1 = X_2$ for clause 1, $X_1 = Y$ and $X_1 = X_2$ for clause 2, and $X_1 = U$ for clause 3. Thus, the minimum partition based on position 1 splits the clauses into two sets: the first consisting of clauses 1 and 2, with the corresponding unification operation $X_1 = X_2$; and the second consisting of clause 3, with the operation $X_1 = U$. Using *part* and *common* functions that partition and identify common operations based on equation sets yields the optimal automaton in Figure 5c. Note that the size of the equation set corresponding to each clause head is quadratic in the number of symbols in the head. Thus, *part*, *common*, and the dynamic programming algorithm still remain polynomial in the worst case.

For nonlinear heads we need to estimate the cost of unifying two positions in the goal. In general this cost is unbounded, since the cost of unifying two positions in a term depends on the sizes of the subterms. Hence, there is no suitable measure for estimating the worst-case cost of an automaton. Nevertheless, if the size of terms is bounded (as in Datalog programs), Theorem 3.3 still holds.

Optimal NSFA

Although Properties 1 and 2 hold in an optimal NSFA, Property 3 does not. In particular, the transitions from a state can partition its compatible clauses into *subsets*. Constructing an optimal NSFA may require enumerating optimal automata for a large number of subsets of clauses for each position associated with a state. In fact, we show:

Theorem 3.4 *The problem of optimal NSFA construction is NP-complete.*

The hardness of finding an optimal factoring automaton is demonstrated by viewing the automaton as an *order-containing full* trie (in the terminology of Comer and Sethi [4]), and showing that the corresponding trie minimization problem is NP-complete. Although in [4] the hardness of constructing minimal full tries and minimal order-containing pruned tries were shown, the hardness of finding a minimal order-containing full trie was left open (the proof appears in the appendix). Note that finding an optimal decision tree [10] corresponds to order-containing pruned trie minimization.

4 Transformation Algorithm

The algorithm used to translate a factoring automaton into Prolog code is described using the programs in Figure 1 and the automaton in Figure 2a as an example. In an automaton, each state with multiple outgoing transitions, called a *branch point*, is associated with a new predicate. When a state is reached, the computation that remains to be done is performed by the associated predicate. The fringe variables of the state are passed as arguments to the predicate, since these represent the positions where remaining computations are to be performed. Furthermore, any head variables in the skeleton are also passed as arguments, since they may be referenced in the clause body. For example, in Figure 2a state s_1 is associated with predicate p with arguments X_1, X_2 and X_3, and state s_2 is associated with predicate p_1 with arguments X_2 and X_3.

Note that the transitions from a branch point represent alternative unifications to be performed. Hence, the associated predicate is defined by a set of clauses, one for each outgoing transition. Each clause performs the sequence of unifications associated with the transitions leading to the next branch point or leaf. In the automaton in Figure 2a the first clause of p needs to perform the unification $X_1 = a$, and thus the head of the clause is $p(a, X_2, X_3)$. If the sequence of transitions ends in a branch point, the body of the clause is a call to the predicate associated with that branch point; otherwise, it is the body of the clause associated with the leaf. In the above example, the body of the first clause of p is $p_1(X_2, X_3)$.

An algorithm to translate any factoring automaton into Prolog code is given in Figure 6. The algorithm is invoked with the start state and the name of the predicate as arguments. The clauses of the translated program are added to the set $\mathcal{P}$ (initially empty). The complete translation of the automaton in Figure 2a appears in Figure 1b. Similarly, the translation of the automaton in Figure 5c appears in Figure 5d.

Observe that choosing the appropriate transition in the automaton corresponds to an indexing operation in the resulting program. If more than one transition is possible at any state, a choice point will be placed during execution of the transformed program. Thus, the operations of recording pending transitions and backtracking through them in the automaton correspond to placing and backtracking through choice points in the execution of the resulting Prolog program.

Transformation in the presence of cuts

In general, unification factoring at the source-level does not preserve the semantics of predicates containing cuts. For example, on query $p(X, Y)$ the program in Figure 7a computes $\langle p(a, b)\rangle$, while the optimal transformed program in Figure 7b returns $\langle p(a, b), p(b, d)\rangle$. This problem arises due to implicit scoping of cuts in Prolog. Specifically, a cut removes the choice point placed when the *current* predicate was called, as well as all subsequent choice points. We say that a cut *cuts to* the most recent choice point of the current predicate. If this choice point can be explicitly specified, then the transformation still preserves the program's semantics. In the XSB compiler, for instance, unification factoring is performed after a transformation that makes the scope of cuts explicit. Figure 7c shows the effect of applying the cut transformation used in XSB to the predicate in Figure 7a. Unification factoring is then applied to the cut transformed program, yielding the predicate in Figure 7d. The semantics of the original predicate is preserved. Thus, in XSB, unification factoring is uniformly applied to all programs, including those that have cuts.

5 Implementation and Performance

A direct execution of the transformed program can introduce unnecessary inefficiences mainly due to increased data movement and procedure calls on newly introduced predicates.

Data movement In the WAM, the arguments of a predicate are stored in WAM registers, where the i^{th} argument is kept in register i. If the i^{th} argument of one predicate is used as the j^{th} argument in a call to another, that argument must be moved from register i to register j. Consider the two clauses in Figure 8a and the result of the transformation (fig. 8b). Each call to $p2/2$ in the body of $p/3$ requires movement of two arguments. Such movement can be reduced in a number of ways, including the use of place-holding arguments (see Figure 8c). However, the potential for reducing data movement is limited by a system's indexing facilities. To index calls in Figure 8c, for example, would require a system, such as XSB, that is able to index on an argument other than the first.

Inlining By inlining the new predicates, calls to them are avoided, reducing execution time. Secondly, since these predicates are not user callable, symbol table size is reduced. Thirdly, since these predicates have only one call site, inlining does not create multiple copies, reducing code space. Finally, inlining restores the call structure of the original program, making the transformation *transparent* to program tracing.

Performance Table 1 shows the effect of performing unification factoring as a source transformation (that includes optimization for data movement) in three different Prolog systems[3]. In that table, the columns labeled 'Speedup' list the ratio of the CPU time taken by each query on the transformed program to that on the original program. The increase in the sizes of object files due to the program transformation are listed under the heading 'Object size increase'. All figures were obtained using standard WAM indexing (first argument, principal functor). The columns labeled 'Inline' illustrate the benefits of inlining, as implemented in the XSB compiler.

Programs `dnf` (Dutch national flag), `LL(k)` (a parser), `border` (from CHAT-80 [13]), and `replace` (an expression translator) and the corresponding queries were taken from [2]. Programs `map` (a map coloring program), `mergesort`,

[3] The systems used were Quintus Prolog Release 3.0, SICStus Prolog 2.1 #9, and XSB version 1.4.0. All benchmarks were run on a SparcStation 2 running SunOS 4.1.1. Benchmark programs can be obtained by anonymous ftp from cs.sunysb.edu in the directory /pub/XSB/benchmarks.

algorithm *translate*(*state*, *pname*)
  let $\{\pi_1, \ldots, \pi_m\}$ be *fringe*(S_{state})
  let $\{Y_1, \ldots Y_k\}$ be the set of head variables (i.e., $\in \mathcal{V}$) in skeleton S_{state}
  foreach edge (*state*, *dest*)
    let *state'* be the first branch point or leaf on the path from *state* through *dest*
    $head \leftarrow pname(Y_1, \ldots, Y_k, t_1, \ldots, t_m)$, where t_i is the subterm of skeleton $S_{state'}$ rooted at π_i
    if *state'* is a branch point
      let *pname'* be a new predicate name
      $body \leftarrow pname'(Y_1, \ldots, Y_k, X_{\pi'_1}, \ldots, X_{\pi'_l})$, where $\pi'_1, \ldots, \pi'_l$ are the fringe positions of $S_{state'}$
      translate(*state'*, *pname'*)
    else /* *state'* is a leaf */
      $body \leftarrow$ body of clause $C_{state'}$
    add (*head* :– *body*) to $\mathcal{P}$

Figure 6: Translation Algorithm

(a)
```
p(a,b) :- !.
p(a,c).
p(b,d).
```

(b)
```
p(a, X) :- p1(X).
p(b, d).
p1(b) :- !.
p1(c).
```

(c)
```
p(X,Y) :- _$savecp(Z),
          _$p(X,Y,Z).
_$p(a,b,X) :- _$cutto(X).
_$p(a,c,_).
_$p(b,d,_).
```

(d)
```
p(X,Y) :- _$savecp(Z),
          _$p(X,Y,Z).
_$p(a,X,Y) :- _$p1(X,Y).
_$p(b,d,_).
_$p1(b,X) :- _$cutto(X).
_$p1(c,_).
```

Figure 7: Original (a), unsound transformation (b), cut transformed (c), factored (d)

(a)
```
p(a,b,c).
p(a,c,d).
```

(b)
```
p(a,X,Y) :- p2(X,Y).
p2(b,c).
p2(c,d).
```

(c)
```
p(a,X,Y) :- p2(_,X,Y).
p2(_,b,c).
p2(_,c,d).
```

Figure 8: Original (a) and transformed predicate before (b) and after (c) data movement reduction

Program [Query]	Speedup				Object size increase			
	Source			Inline	Source			Inline
	Quintus	SICStus	XSB	XSB	Quintus	SICStus	XSB	XSB
dnf [s1]	2.12	1.32	1.99	2.48	1.07	1.03	1.06	1.01
dnf [s2]	2.05	1.33	1.89	2.39				
dnf [s3]	1.83	1.40	1.68	2.03				
LL(1) [p]	0.83	0.59	1.10	1.16	1.13	1.17	1.13	1.01
LL(2) [q]	1.18	1.16	1.32	1.40	1.12	1.14	1.08	1.02
LL(3) [r]	1.22	1.21	1.54	1.63	1.17	1.14	1.08	1.00
border [medit.]	2.00	1.29	2.11	2.38	1.05	1.04	0.99	0.93
border [hungary]	1.58	1.14	1.64	1.84				
border [albania]	1.08	1.00	1.03	1.16				
replace.sw [neg]	0.82	0.86	0.97	0.97	1.00	1.05	0.98	0.96
replace.sw [<]	0.77	0.82	0.91	0.93				
replace.sw [mul]	0.96	0.96	0.98	1.00				
Synchem [alcohol]	2.16	1.96	1.99	2.27	1.62	1.47	1.37	1.04
Synchem [ether]	2.65	2.33	2.93	3.55				
Synchem [cc_dbl]	2.71	2.53	2.67	2.97				
map	1.51	1.50	1.17	1.36	1.18	1.19	1.25	1.09
mergesort	0.99	0.97	1.32	1.41	1.10	1.07	1.05	1.00
mutest	0.96	0.93	0.99	1.00	1.07	1.05	1.10	1.01
isotrees (linear)	0.83	0.83	1.11	1.18	1.09	1.04	1.09	0.98
isotrees (non-linear)	0.99	0.93	1.14	1.23	1.05	1.03	1.00	0.92

Table 1: Speedups and object size increases for unification factoring

Program [Query]	Speedup							
	Source						Inline	
	Quintus		SICStus		XSB		XSB	
	Before	After	Before	After	Before	After	Before	After
`LL(1)` [p]	0.86	1.30	0.61	1.11	1.09	1.49	1.16	1.57
`replace.sw` [neg]	0.82	3.60	0.86	3.22	0.97	4.27	0.97	4.35
`replace.sw` [<]	0.77	2.65	0.82	2.10	0.91	2.58	0.93	2.65
`replace.sw` [mul]	0.96	1.20	0.96	1.21	0.98	1.09	1.00	1.13

Table 2: Speedups before and after mode optimization

and `mutest` (a theorem prover) are benchmarks from the Andorra system. The `Synchem` benchmarks are queries on a 5,000-fact chemical database. The `isotrees` program illustrates the effect of sharing non-linear unifications.

Based on the performance results we first summarize the strengths of unification factoring. Factoring of match operations in structures results in improved indexing and hence in performance, *e.g.,* `dnf`, `LL(2)`, `LL(3)`, and `map`. Similarly factoring on shared arguments as in `border` improves performance. Speedups in `mergesort` and `isotrees` are due essentially to sharing of computed substitutions. The isotrees example also illustrates the benefits of factoring non-linear unification. In programs such as `Synchem`, where both match operations and computed substitutions are shared, the performance gains are much more significant. Finally, unification factoring does not degrade performance in the absence of sharable operations, as in the `mutest` example.

The XSB speedups are generally larger than those for Quintus and SICStus, even when inlining is not performed. XSB's engine, which supports restricted SLG resolution, requires more expensive trailing, and untrailing, than the WAM. The number of these operations is reduced by unification factoring. Parallel Prologs such as Andorra have similar expenses so that unification factoring can be expected to provide similar speedups for such systems.

The transformation increases the size of the object files only by small amounts. The largest size increase is for the `Synchem` database, and is mainly the result of the increase in symbol table size due to the new predicates introduced by the transformation. But notice that by inlining even this has been substantially reduced.

Programs `LL(1)` and `replace.sw` show a slowdown primarily due to factoring unifications on an output argument, resulting in loss of indexing. Recall that an optimal factoring automaton is found assuming no knowledge of the goal. When modes are known, we could first build a complete switching tree for all the input-moded arguments (as in [6]), and then attach optimal SFAs for the unmoded arguments at the leaves of the switching tree. Note, however, that the space of a switching tree can be exponential in the worst case. Hence we use the following simple technique of building SFAs in the presence of modes. We divide the fringe positions into input-moded positions and unmoded positions, and all input-moded positions are inspected before any unmoded position is inspected. The effectiveness of this technique is indicated in Table 2.

6 Discussion

Efficient handling of non-determinism, traditionally ignored by indexing methods, is one of the key strengths of unification factoring. To reflect non-determinism, a factoring automaton makes all possible transitions at each state. In contrast, pattern matching tries and decision trees make at most one transition from any state. Although functional programming languages such as ML use textual order for pattern matching just as Prolog does for unification, there are two notable differences. First of all, the semantics of functional languages require that only one of potentially several matching patterns be selected, whereas in Prolog the clauses of all heads unifying with a goal may be evaluated. Secondly, pattern matching is unidirectional, whereas unification is bidirectional. Thus, the optimality criteria for factoring automata differ substantially from those of the other two structures, necessitating the new techniques developed in this paper for constructing optimal automata.

Our experimental results show that unification factoring is a practical technique that can achieve substantial speedups for logic programs, while requiring no changes in the WAM and resulting in virtually no increase in code size. Furthermore, the speedups obtained by the source-to-source transformation on all three Prolog systems are comparable to those obtained by the indexing technique of [2] that involved extensive compiler and WAM modifications.

The speedups observed may be even more substantial when unification factoring is applied to programs which are themselves produced by transformations. For instance, the HiLog transformation [3] increases the declarativeness of programs by allowing unification on predicate symbols. If implemented naively, however, HiLog can cause a decrease in efficiency for clause access. Experiments have shown that unification factoring can lead to speedups of 3 to 4 on HiLog code. Given the demonstrable performance of unification factoring and its simplicity to implement, it is reasonable to expect that unification factoring may become a fundamental tool for logic program compilation.

Unification factoring has been incorporated in the XSB logic programming system, which is available by anonymous ftp from `cs.sunysb.edu` in directory `/pub/XSB`.

References

[1] Leo Bachmair, Ta Chen, and I. V. Ramakrishnan. Associative-commutative discrimination nets. In *Theory and Practice of Software Development*, number 668 in LNCS, pages 61–74. Springer Verlag, April 1993.

[2] T. Chen, I. V. Ramakrishnan, and R. Ramesh. Multistage indexing algorithms for speeding Prolog execution. In *Joint International Conference/Symposium on Logic Programming*, pages 639–653, 1992.

[3] W. Chen, M. Kifer, and D. S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[4] D. Comer and R. Sethi. The complexity of trie index construction. *Journal of the ACM*, 24(3):428–440, July 1977.

[5] W. Hans. A complete indexing scheme for WAM-based abstract machines. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 232–244, 1992.

[6] T. Hickey and S. Mudambi. Global compilation of Prolog. *Journal of Logic Programming*, 7:193–230, 1989.

[7] S. Kliger and E. Shapiro. From decision trees to decision graphs. In *North American Conference on Logic Programming*, pages 97–116, 1991.

[8] D. Palmer and L. Naish. NUA-Prolog: An extension to the WAM for parallel Andorra. In *International Conference on Logic Programming*, pages 429–442, 1991.

[9] R. Ramesh, I. V. Ramakrishnan, and D. S. Warren. Automata-driven indexing of Prolog clauses. In *ACM Symposium on Priciples of Programming Languages*, pages 281–290. ACM Press, 1990.

[10] R. L. Rivest and L. Hyafil. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, May 1976.

[11] R. C. Sekar, I. V. Ramakrishnan, and R. Ramesh. Adaptive pattern matching. In *International Conference on Automata, Languages, and Programming*, number 623 in LNCS, pages 247–260. Springer Verlag, 1992. To appear in *SIAM J. Comp.*

[12] E. Tick and M. Korsloot. Determinacy testing for nondeterminate logic programming languages. *ACM Transactions on Programming Languages and Systems*, 16(1):3–34, January 1994.

[13] D. H. D. Warren and F. C. N. Pereira. An efficient easily adaptable system for interpreting natural language queries. *American Journal of Computational Linguistics*, 8(3-4):110–122, 1982.

[14] N. Zhou, T. Takagi, and K. Ushijima. A matching tree oriented abstract machine for Prolog. In *International Conference on Logic Programming*, pages 159–173. MIT Press, 1990.

A Appendix

A.1 Optimal SFA construction

Construction of an optimal SFA for the sequence of clauses $\langle p(a,b,c), p(a,b,d), p(a,c,c), p(b,a,c)\rangle$ defining the predicate in Figure 9a begins with the computation of its cost, using Equation 1. The cost and root position of the lowest cost automaton computed for a subsequence with end points (i,i') at any point in the computation is stored in a table (Fig. 9b) at entry (i,i'), where i is the row and i' the column.

We begin by finding positions having symbols common to all four clauses:

$$common(p(X_1,X_2,X_3),1,4) = (\{\}, p(X_1,X_2,X_3)).$$

There are no common positions, so any of positions \$1, \$2, and \$3 might be used for the root state. We first try position 1, and compute the partition:

$$part(1,4,\$1) = \{(a,1,3),(b,4,4)\}.$$

We now need to compute optimal sub-automata for subsequences $\langle p(a,b,c), p(a,b,d), p(a,c,c)\rangle$ and $\langle p(b,a,c)\rangle$

Repeating the above process for subsequence $(1,3)$,

$$common(p(X_1,X_2,X_3),1,3) = (\{(\$1,a)\}, p(a,X_2,X_3))$$

shows that subsequence $(1,3)$ has one common position (\$1) leaving positions \$2 and \$3 as possible root positions for the sub-automaton. We first choose position \$2:

$$part(1,3,\$2) = \{(b,1,2),(c,3,3)\}$$

Continuing similarly for subsequence $(1,2)$ gives

$$\begin{aligned} common(p(a,X_2,X_3),1,2) &= (\{(\$2,b)\}, p(a,b,X_3)) \\ part(1,2,\$3) &= \{(c,1,1),(d,2,2)\} \end{aligned}$$

$$\begin{aligned} common(p(a,b,X_3),1,1) &= (\{(\$3,c)\}, p(a,b,c)) \\ common(p(a,b,X_3),2,2) &= (\{(\$3,d)\}, p(a,b,d)) \end{aligned}$$

Now, each subsequence is a single clause. Since all positions in a single clause are common positions, no positions for further partitioning are available, and the cost for each single clause is 0. Thus, the cost of the sub-automaton for subsequence $(1,2)$ rooted at \$3 is 2 (one each for transitions *unify*$(\$3,c)$ and *unify*$(\$3,d)$). Cost 2 and position 3 are then stored in entry $(1,2)$ of the table.

Returning to compute the cost for subsequence $(3,3)$,

$$common(p(a,X_2,X_3),3,3) = (\{(\$2,c),(\$3,c)\}, p(a,c,c))$$

gives the cost of the sub-automaton for subsequence $(1,3)$ with root position \$2 as $1+2+2=5$ (one for transition *unify*$(\$2,b)$; one each for transitions *unify*$(\$2,c)$ and *unify*$(\$3,c)$; and two as computed for subsequence $(1,2)$. Thus, cost 5 and position 2 are stored in entry $(1,3)$ of the table.

Completing the computation for sequence $(1,4)$ at position \$1 yields additional costs of one (for the transition labeled *unify*$(\$1,a)$) and three (for transitions *unify*$(\$1,b)$, *unify*$(\$2,a)$, and *unify*$(\$3,c)$ for subsequence $(4,4)$), giving a total cost of 9. Thus, cost 9 and position 1 are stored in entry $(1,4)$ of the table (highlighted). The automaton corresponding to this cost and position is shown in Figure 9c. Verifying that no other choice of position yields a better automaton is left to the interested reader. Note that the shaded entries in the table are not used in computing the cost of an optimal automaton.

A.2 NP-completeness of optimal NSFA construction

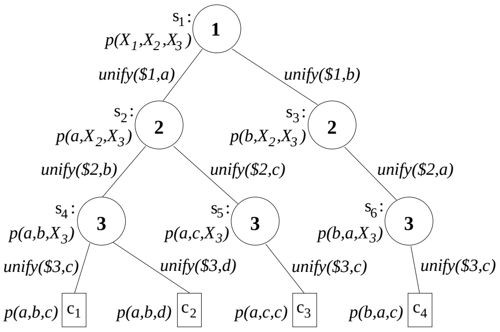
A non-sequential factoring automaton can be viewed abstractly as a trie, in which the clause heads of a predicate are viewed as strings of symbols. Optimization of an NSFA (when elementary unification operations are assumed to have unit cost) thus corresponds to trie minimization. In the terminology of Comer and Sethi [4], an NSFA corresponds to a full order-containing trie (full O-trie), where "full" refers to the fact any root-to-leaf path in the trie examines an entire string, and "order-containing" means that different paths may examine characters (positions) in different orders.

```
p(a,b,c).
p(a,b,d).
p(a,c,c).
p(b,a,c).
```

(a)

i \ i'	1	2	3	4
1	Cost: 0 Pos: -	Cost: 2 Pos: 3	Cost: 5 Pos: 2	Cost: 9 Pos: 1
2		Cost: 0 Pos: -		
3			Cost: 0 Pos: -	Cost: 2 Pos: 1
4				Cost: 0 Pos: -

(b)



(c)

Figure 9: Optimal SFA construction: predicate (a), cost table (b), SFA (c)

Theorem A.1 *The minimization problem for full O-tries (FOT) is NP-complete.*

Proof: We show that the trie minimization problem is NP-hard by reduction from the minimum set cover problem (SC). The minimum set cover problem can be stated as follows: Given a finite set $U = \{u_1, \ldots, u_n\}$, a collection $C = \{C_1, \ldots, C_m\}$ of subsets of U, and a positive integer $k \leq m$, do there exist k or fewer subsets in C whose union is S?

Let I_{SC} be an instance of SC. We construct an instance I_{FOT} of FOT with $2n$ strings, each of length $2(n+m)^2+m$, as follows. Each string consists of three fields: a "*Test*" field, consisting of m characters; a "*Blue*" field, consisting of $(n+m)^2$ characters; and a "*Red*" field, consisting of $(n+m)^2$ characters:

$$\underbrace{T_1 T_2 \cdots T_m}_{Test}\underbrace{B_1 B_2 \cdots B_{(n+m)^2}}_{Blue}\underbrace{R_1 R_2 \cdots R_{(n+m)^2}}_{Red}$$

For each element $u_i \in U$ two strings are constructed: a *Red* string and a *Blue* string. In the *Red* string, $T_j = 0$, for $1 \leq j \leq m$; $B_l = i$ and $R_l = 0$, for $1 \leq l \leq (n+m)^2$. The i^{th} *Red* string thus has the form

$$\underbrace{0\cdots0}_{Test}\underbrace{i\cdots i}_{Blue}\underbrace{0\cdots0}_{Red}$$

In the *Blue* string, $T_j = 1$ if $u_i \in C_j$, otherwise 0, for $1 \leq j \leq m$; $B_l = i$ and $R_l = 0$, for $1 \leq l \leq (n+m)^2$. The i^{th} *Blue* string thus has the form

$$\underbrace{T_1 T_2 \cdots T_m}_{Test}\underbrace{0\cdots0}_{Blue}\underbrace{i\cdots i}_{Red}$$

Checking a character in the *Test* field of a *Blue* string can be thought of testing for membership of an element of U in a subset.

Observe that the *Red* and *Test* fields in every *Red* string are identical, and that each *Blue* field is distinct. For a set consisting entirely of *Red* strings, a minimal trie must test all characters in the *Red* and *Test* fields before testing any character in the *Blue* field, since the first test in the *Blue* field effectively partitions the set into individual strings (see Figure 10a). The order of testing within the *Red* and *Test* fields is unimportant, as is the order of testing within the *Blue* field.

Also observe that the *Blue* field in every *Blue* string is identical, and that each *Red* field is distinct. A minimal trie for a set of *Blue* strings must test all characters in the *Blue* field before testing any character in the *Red* field. In general, testing characters in the *Test* field will incrementally partition the set. Thus, a minimal trie for *Blue* strings will typically have the form shown in Figure 10b.

The above observations lead to the following bounds on the sizes of minimal tries for *monochromatic* sets of strings.

Lemma A.1.1 *Given a set S consisting entirely of Red strings, such that $|S| = n_R$, the number of edges in a minimal full O-trie for S is exactly $(n_R+1)(n+m)^2+m$.*

Lemma A.1.2 *Given a set S consisting entirely of Blue strings, such that $|S| = n_B$, the number of edges in a minimal full O-trie for S is no more than $(n_B+1)(n+m)^2+mn_B$.*

The main idea in constructing a minimal full trie is to order the tests such that less partitioning occurs near the root and more occurs toward the leaves. Therefore, to build a minimal trie for the $2n$ *Blue* and *Red* strings constructed from I_{SC}, testing in the *Test* field should precede testing in the *Blue* and *Red* fields. Observe that testing a character in the *Test* field partitions a set containing both *Blue* and *Red* strings into one set containing only *Blue* strings (on a branch labeled "1") and another containing *Red* and perhaps some *Blue* strings (on a branch labeled "0"– see Figure 11a). Testing a character in the *Red* (*Blue*) field, on the other hand, partitions the set into one set containing all of the *Red* (*Blue*) strings and one set for each of the *Blue* (*Red*) strings (see Figure 11b). These observations lead to the following proof that U has a cover of size k if and only if there exists a full O-trie for S having fewer than $(2n+k+2)(n+m)^2$ edges.

Lemma A.1.3 *If U has a cover of size k, then there exists a full O-trie over S having fewer than $(2n+k+2)(n+m)^2$ edges.*

Proof of lemma: A trie over S can be constructed as illustrated in Figure 11a. For $1 \leq j \leq k$, let C_{i_j} be a member of the cover of U. Now, for each node T_{i_j} in the trie, the subtrie attached to the "1" edge is a trie over *Blue* strings, since no *Red* string contains a 1 in its T field. Since $\{C_{i_1}, \ldots, C_{i_k}\}$ is a cover for U, each *Blue* string is represented in one of the subtries attached to a "1" edge, and only the n *Red* strings

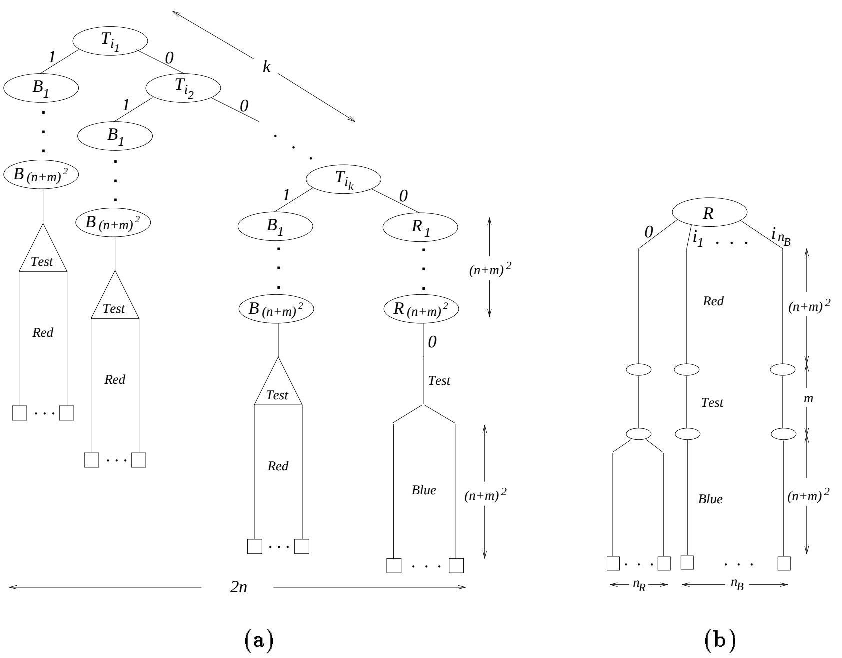


Figure 11: Minimal trie for $2n$ strings (a) and suboptimal trie (b)

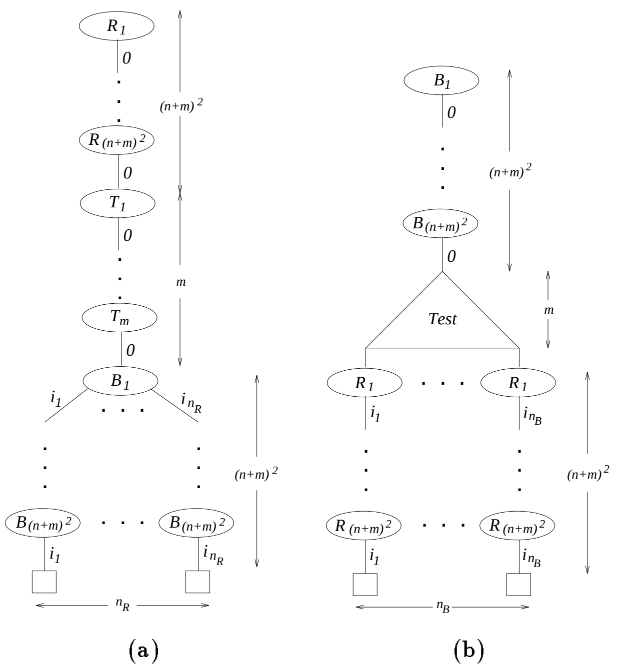


Figure 10: Minimal tries for sets of *Red* (a) and *Blue* (b) strings

are represented in the subtrie attached to the "0" edge of node T_{i_k}.

By lemma A.1.2, the number of edges in the subtrie attached to the "1" edge of node T_{i_j} is no more than $(n_{s_j}+1)(n+m)^2+(m-j)n_{s_j}$, where n_{s_j} is the number of strings represented in the subtrie. Since $\sum_{j=1}^{k} n_{s_j} = n$, the total number of edges in the *Blue* subtries is no more than $(n+k)(n+m)^2+mn$. By lemma A.1.1, the number of edges in the subtrie attached to the "0" edge of node T_{i_k} is no more than $(n+1)(n+m)^2+m-k$. Thus, the total number of edges in the trie is no more than $(2n+k+1)(n+m)^2+(n+1)m+2k$, which is less than $(2n+k+2)(n+m)^2$. □

Lemma A.1.4 *If there exists a full O-trie over S having fewer than $(2n+k+2)(n+m)^2$ edges, then U has a cover of size k.*

Proof of lemma: It suffices to show that any trie over S having fewer than $(2n+k+2)(n+m)^2$ edges must be of the form shown in Figure 11a. Given a set S' of strings containing n_R *Red* strings and n_B *Blue* strings, the root of a minimal trie over S' must be a T node. If, instead, the root were a R node, the trie would have at least $(2n_B+n_R+1)(n+m)^2$ edges (see Figure 11b). Similarly, a trie with a B root node would have at least $(2n_R+n_B+1)(n+m)^2$ edges. Replacement of any of the k T nodes in the trie in Figure 11a by a R or B node would therefore result in at least $(n-k)(n+m)^2$ additional edges. Thus, any trie having fewer than $(2n+k+2)(n+m)^2$ edges must be of the form shown in Figure 11a, which can exist only if U has a cover of size k. □

Lemmas A.1.3 and A.1.4 together complete the reduction from I_{SC} to I_{FOT}. The remaining details showing that the transformation is polynomial in the size of I_{SC} and that $FOT \in NP$ are standard, and are omitted. ■