Adaptive Pattern Matching[†]

(Submitted to SIAM J. Computing)

R.C. Sekar[‡]
Bellcore, Rm 2A-274
445 South Street
Morristown, NJ 07962
sekar@thumper.bellcore.com

R. Ramesh[§]
Dept. of Computer Science
University of Texas at Dallas
Richardson, TX 75083
ramesh@utdallas.edu

I.V. Ramakrishnan[¶]
Dept. of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794
ram@cs.sunysb.edu

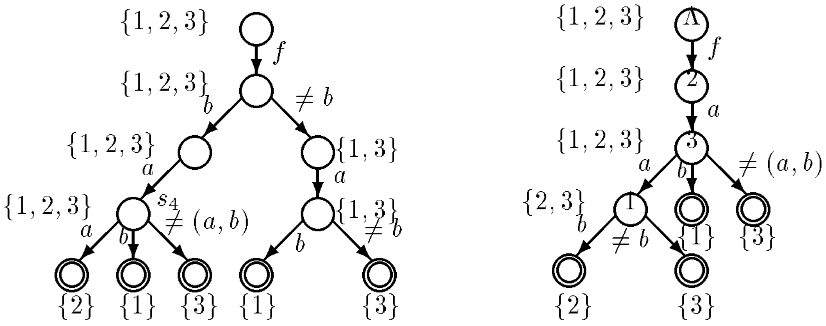
Abstract

Pattern matching is an important operation used in many applications such as functional programming, rewriting and rule-based expert systems. By preprocessing the patterns into a DFA-like automaton, we can rapidly select the matching pattern(s) in a single scan of the relevant portions of the input term. This automaton is typically based on left-to-right traversal of the patterns. By *adapting* the traversal order to suit the set of input patterns, it is possible to considerably reduce the space and matching time requirements of the automaton. The design of such adaptive automata is the focus of this paper. We first formalize the notion of an adaptive traversal. We then present several strategies for synthesizing adaptive traversal orders aimed at reducing space and matching time complexity. In the worst case, however, the space requirements can be exponential in the size of the patterns. We show this by establishing an exponential lower bounds on space that is *independent* of the traversal order used. We then discuss an orthogonal approach to space minimization based on *direct* construction of optimal dag automata. Finally, our work brings forth the impact of typing in pattern matching. In particular, we show that several important problems (e.g., lazy pattern matching in ML) are computationally hard in the presence of type disciplines, whereas they can be solved efficiently in the untyped setting.

1 Introduction

Pattern matching is a fundamental operation in a number of important applications such as functional and equational programming [11, 21], term rewriting, theorem proving [7] and rule-based systems[6]. In most of these applications, patterns are partially ordered by assigning priorities. For instance, in languages such as ML [10] and Haskell [13], a pattern occurring earlier in the text has a higher priority over those following it. In HOPE [3] and in many rule-based systems, more specific patterns have higher priority over less specific ones [6, 20]. In theorem proving applications, priorities are used to encode efficient heuristics. Applications that do not impose priorities can be handled as a special case, by assigning equal priorities to all patterns.

[†]A preliminary version of this paper appears in proceedings of the ICALP, 1992.
[‡]Research completed at SUNY, Stony Brook and supported by NSF grants CCR-8805734,9102159 and NYS S&T grant RDG 90173.
[§]Research supported by NSF grant CCR-9110055.
[¶]Research supported by NSF grants CCR-8805734,9102159 and NYS S&T grant RDG 90173.



Left-to-right automaton

Adaptive automaton

Priority	Pattern
1	$f(x, a, b)$
2	$f(b, a, a)$
3	$f(x, a, y)$

variables: x, y
nonvariables: f, a, b

Figure 1: Automata for $f(x, a, b), f(b, a, a), f(x, a, y)$

The typical approach to fast pattern matching is to preprocess the patterns into a DFA-like automaton that can rapidly select the patterns that match the input term. The main advantage of such a matching automaton is that all pattern matches can be identified in a *single* scan (i.e., no backtracking) of portions of input term relevant for matching purposes and is done in time that is *independent* of the number of patterns. Existing automaton-based approaches (as well as the approach we present in this paper) do not handle non-linear patterns (i.e., patterns with multiple occurrences of the same variable). This is because many applications use only linear patterns; and even for those applications that permit non-linear patterns, it has been observed that most failures are associated with symbol mismatches [4], which can be detected without taking non-linearity into account.

Figure 1 shows a matching automaton constructed on the basis of a left-to-right traversal of patterns. Each state of the automaton corresponds to the prefix of the input term seen in reaching that state and is annotated with the set of patterns that can possibly match. For instance, state s_4 corresponds to having inspected the prefix $f(b, a, x)$, where x denotes the subterm that has not yet been examined. This state is annotated with the pattern set $\{1, 2, 3\}$ since we cannot rule out a match for any of the three patterns on the basis of the prefix $f(b, a, x)$.

Pattern matching automata have been extensively studied for well over a decade. Christian [4] obtained dramatic speedups in a Knuth-Bendix completion system by using an automaton for unprioritized patterns based on left-to-right traversal. Graf [9] also describes a similar automaton for unprioritized patterns. In functional programming, Augustsson [1, 2] and Wadler [26] describe pattern matching techniques that are also based on left-to-right traversal, but do deal with priorities.

The methods of Augustsson and Wadler are economical in terms of space usage, but may reexamine symbols in the input term. In the worst case, these methods can degenerate to the naive method of testing each pattern separately against the input term. In contrast, the methods of Christian [4], Graf [9], and Schnoebelen [24] avoid reexamining symbols, but this is achieved at the cost of increasing the space requirements. In fact, Graf and Schnoebelen show that the upper bounds on the size of their automata are exponential.

One way to improve both space *and* matching time is to engineer a traversal order to suit the set of patterns or the application domain. We refer to such traversals as *adaptive traversals* and automata based on such traversals as *adaptive automata*. As traversal orders are no longer fixed a priori, an adaptive automaton must specify the traversal order. For instance, in the adaptive automaton shown in Figure 1, each state is labeled with the next argument position to be inspected from that state. Adaptive traversal has the following advantages over a *fixed-order* of traversal such as left-to-right.

- The resulting automaton is usually smaller, e.g., the adaptive automaton in Figure 1 has 8 states compared to 11 in the left-to-right automaton. The reduction factor can even become *exponential.*
- Pattern matching requires lesser time with adaptive traversals than fixed-order traversals. For example, the left-to-right automaton needs to inspect four symbols to announce a match for pattern 1. The adaptive automaton inspects only a subset of these positions (three of them) to announce the match. Examining unnecessary symbols is especially undesirable in the context of lazy functional languages since it runs counter to the goals of lazy evaluation.

Unlike automata based on fixed order of traversals, relatively very little is known about designing automata based on adaptive traversals. There are a number of interesting questions that arise in such a design. For instance, given a set of patterns how do we choose a traversal order to realize the advantages mentioned above? Is it possible to select a traversal order that improves space and matching time over automata based on fixed-traversal orders? What are the lower and upper bounds on space and matching time complexities of such automata? Finally, in the context of functional programming, using arbitrary order of traversal for pattern matching is inappropriate as it affects the termination properties of the program. Given this constraint, is it possible to internally change the traversal order in such a way that it does not affect the termination properties and at the same time realize the advantages of adaptive traversal? We answer all these questions in this paper. In addition, we also solve some of the problems that have remained open even in the context of left-to-right traversals. Such problems include

- Tight bounds on space and matching time complexity
- Impact of type discipline on the computational complexity of many problems that arise in construction of matching automata
- Direct construction of optimal automata that shares equivalent states.

In the following section, we summarize our results.

1.1 Summary of Results

In section 2, we first formalize the concept of adaptive traversal and its special case, fixed traversal orders. We then present a generic algorithm for building an adaptive automaton that is parameterized with respect to the traversal order. Our generic algorithm is used as a basis for the results presented in later sections. As mentioned before, our algorithms assume that the patterns are linear, i.e., no variable in any pattern occurs more than once. The automaton can still be used for non-linear patterns, but on reaching a final state, we would have to check for consistency of the substitutions for the different occurrences of the same variable.

In section 3, we present several techniques to synthesize traversal orders that can improve space and matching time:

- We develop the important concept of a *representative set* which forms the basis of several optimization techniques aimed at avoiding inspection of unnecessary symbols.
- We present several powerful strategies for synthesizing traversal orders. Through an intricate example, we show that they all can sometimes increase both space and matching time.
- We then show that a strategy that inspects *index* positions whenever possible, does not suffer from the above drawback. Specifically, we show that inspecting index positions can only improve space and matching time of the automata. Huet and Lévy[14] had established the importance of index in the design of an optimal automaton (in the sense of not seeing any unnecessary symbols) for strongly-sequential patterns. Our results extend the applicability of indices even for patterns that are not strongly-sequential.
- Finally, we study synthesis of traversal orders that are appropriate for lazy functional languages. In such languages, pattern-matching is closely coupled with evaluation in such a way that the traversal order used for matching can affect the termination properties of the program. Therefore the programmer must be made aware of the traversal order T used for matching. The question then is whether we can synthesize a traversal order that has the same (or better) termination properties as T. In section 3.5, we show how to synthesize such a traversal $S(T)$ that enables the programmer to assume T whereas an implementation can benefit from significant improvements in space and time using the adaptive traversal $S(T)$.

In section 4, we discuss the computational aspects of the strategies presented in section 3. Our work clearly brings forth the impact of typing in pattern matching. We have shown that several important problems in the context of pattern matching are unlikely to have efficient algorithms in typed systems whereas we give polynomial time algorithms for them in untyped systems. In particular:

- We present a quadratic-time algorithm for computing representative sets in untyped systems whereas we show that computing these sets is NP–complete for typed systems.
- In section 4.2 we focus on the important problem of index computation in prioritized systems. Laville [18], Puel and Suarez [23] have extended Huet-Lévy's [14] index computation algorithm to deal with priorities. However, all these algorithms require exponential time in the worst case. In contrast, we show that indexes can be computed in polynomial-time in the case of untyped systems.
- We also show that index computation in typed systems is *co-NP*−complete.

In section 5, we examine space and matching time complexity of the adaptive automata. We show that the space requirements of the automata can be quite large by establishing the first known tight exponential lower bounds on size.

In section 6, we describe an orthogonal approach to space minimization based on directed acyclic graph (dag) representation. By tightly characterizing the equivalent states of the adaptive automata, we directly build their optimal dag representation. Since the unoptimized automaton can be exponentially larger than the optimized one, a direct construction can use polynomial space and time whereas a method that uses FSA minimization techniques may require exponential time and space. As mentioned in [9], this important problem of direct construction had remained open even for left-to-right traversals.

Finally, we conclude the paper in section 7 with a discussion of the various strategies presented in the paper and how they can be combined to build efficient adaptive automata.

2 Preliminaries

In this section we develop the notations and concepts that will be used in the rest of this paper. We also present a generic algorithm for construction of an adaptive automaton. This generic algorithm forms the basis for the results presented in later sections.

We assume familiarity with the basic concept of a *term*. The symbols in a term are drawn from a non-empty *ranked alphabet* Σ and a countable set of variables $\mathcal{X}$. (The term *arity* is sometimes used in the literature in place of *rank*.) We will use a, b, c, d and f to denote nonvariable symbols and x, y and z (with or without subscripts and primes) to denote variables. We use '_' to denote unnamed variables. Each occurrence of '_' denotes a distinct variable. Let $root(t)$ denote the symbol appearing at the root of a term t. In order to refer to subterms of a term, we develop the following concept of a *position*[1]:

Definition 1 (Position) *A position is either*

- *the empty string Λ that reaches the root of the term, or*
- *$p.i$, where p is a position and i an integer, which reaches the i^{th} argument of the root of the subterm reached by p.*

If p is a position then the subterm of t reached by p is denoted by t/p. We use $t[p \leftarrow s]$ to denote the term obtained from t by replacing t/p by s. The set of variable positions in a term t is known as the *fringe* of t. We illustrate these concepts using the term $t = f(a(x), b(a(y), z))$. Here, $t/\Lambda = t, t/2 = b(a(y), z), t/2.1 = a(y)$ and $t/2.2 = z$. The term $t[2 \leftarrow c] = f(a(x), c)$ is obtained by replacing the second argument of f by (the term) c. The fringe of t is $\{1.1, 2.1.1, 2.2\}$.

A *substitution* is a mapping from variables to terms. Given a substitution β, an *instance* $t\beta$ of t is obtained by substituting $\beta(x)$ for every variable x in t. If t is an instance of u then we say $u \leq t$ and call u a *prefix* of t. The inverse of $\leq$ relation is denoted by $\geq$. For the term $t = f(a(x), b(y, z))$ and the substitution β that maps x to $b(x', x''), y$ to c and z to itself, $t\beta = f(a(b(x', x'')), b(c, z))$.

We say that two terms t and s unify, denoted $t \uparrow s$, iff they possess a common instance. $t \sqcup s$ denotes the least such instance in the ordering given by $\leq$. For example, the terms $t = f(a(x), y)$ and $s = f(x', b(y', z'))$ possess common instances $f(a(x''), b(y'', z''))$, $f(a(c), b(y'', z''))$, $f(a(x''), b(a(y''), z''))$ etc. The least common instance $t \sqcup s$ is the term $f(a(x''), b(y'', z''))$.

In the rest of the paper, we use $\mathcal{L}$ to denote the given set of patterns. (A *pattern* is simply a term.) All patterns are assumed to be *linear*, i.e., no variable in them appears more than once. Given such a set and the priority relationship among the patterns, we formalize the notion of pattern matching as follows:

[1]The terminology *occurrence* and *path* are sometimes used in the literature to denote the same concept.

Definition 2 (Pattern Match) *A pattern $l \in \mathcal{L}$ matches u iff $l \leq u$, and no $l' \in \mathcal{L}$ with priority greater than that of l unifies with u. If there is any $l \in \mathcal{L}$ that matches u then we say that there is a* pattern match *for u.*

Note that this definition differs from the traditional notion of matching in that it takes priorities into account. The traditional notion only requires that the term u be an instance of the pattern l. Here, we also require that u not unify with a pattern of higher priority. The intuition behind this requirement is that u actually denotes a prefix of the term t that is being inspected to identify a match[2]. If it unifies with a pattern of higher priority, then we may identify a match for this higher priority pattern when we inspect some of the symbols below this prefix. Since we cannot rule out a match for any higher priority pattern without inspecting the symbols below u, we cannot declare a match for l. Also note that, by definition of pattern match, when we do identify a match for u, we can announce a match for t itself.

To illustrate the concept of a pattern match, consider once again the set of patterns in Figure 1. Only the first pattern matches the term $f(a, a, b)$. Note that no pattern matches the term $f(x, a, a)$ although it is an instance of the third pattern. This is because $f(x, a, a)$ unifies with the second pattern which has a higher priority over the third one.

Given a term u, we define its *match set* $\mathcal{L}_u$, as the set of patterns that unify with u. Observe that by definition of pattern match, we can restrict ourselves to $\mathcal{L}_u$ if we are looking for a match for any term with a prefix u. Furthermore, to identify a match, we can restrict ourselves to inspection of only those fringe positions $\mathcal{F}_u$ wherein at least one of the patterns in $\mathcal{L}_u$ has a nonvariable. In this context, some of the standard traversal orders such as depth-first and breadth-first will also be modified to skip fringe positions that are not in $\mathcal{F}_u$. For illustration of these concepts, consider $\mathcal{L} = \{f(x, a, b), f(x, a, a), f(x, a, y)\}$ and $u = f(x', y', a)$. In this case, $\mathcal{L}_u = \{f(x, a, a), f(x, a, y)\}$. Note that the only variable position in u where every pattern in $\mathcal{L}_u$ has a variable is position 1, and so $\mathcal{F}_u = \{2\}$.

In our search for efficient pattern-matching algorithms, we will need the following concept of an index position which is a fringe position that *must* be inspected by any matching algorithm to announce a match for a term. Formally,

Definition 3 (Index) *A fringe position p for a prefix u is said to be an* index *with respect to a pattern l iff for every term $t \geq u$ such that l matches t, t/p is a nonvariable. It is said to be an index of u w.r.t. a set of patterns $\mathcal{L}$ iff it is an index w.r.t. every $l \in \mathcal{L}$.*

The notion of an index is closely related to the concept of sequentiality [15, 14]. One simple example of an index is a fringe position of u where every pattern in $\mathcal{L}_u$ has a nonvariable. Observe that such a position must be inspected to determine if the given term is an instance of any of the patterns in $\mathcal{L}_u$.

We now classify patterns depending on how they affect the complexity of various problems related to pattern matching. We first classify based on whether more than one pattern can match any given term.

[2] The variables in u denote unexamined positions in t. We deliberately blur the distinction between a variable and uninspected portions of a term. This is because, in linear terms, a variable simply stands for any arbitrary term or unknown term. This correctly reflects our intuition that we have no knowledge of the term structure below a variable in the prefix u.

Definition 4 (Ambiguous and Unambiguous Patterns) *A set of patterns $\mathcal{L}$ is said to be* ambiguous *whenever there is a term t such that more than one pattern matches it. Otherwise $\mathcal{L}$ is* unambiguous.

Our next classification is based on type discipline. In typed systems, the set of allowable input terms are constrained by a type discipline. From a pattern-matching perspective, this constraint takes the form that arguments to a function symbol f must be drawn from a specific set of terms, say, $\{T, F\}$. In this case, terms $f(T, F), f(F, T), f(T, T)$ and $f(F, F)$ are permissible (or well-typed) whereas $f(2, T)$ is not. In untyped systems, there are no restrictions on the arguments of functions. For instance, the terms $f(2, F)$ and $f(c, d)$ are valid input terms. (A more rigorous treatment of types can be readily found in the literature, but is not developed here since the results presented in this paper can be established based only on the simple distinction given here.) The time and space complexity of many problems discussed in this paper will vary widely based on whether the system is typed and also on whether it is ambiguous.

Next we define the notion of a traversal order. We call a traversal order *top-down*, if it visits a node only after visiting its ancestors. Since we are interested only in identifying matches at the root of the input term, we deal with only top-down traversals and not concern ourselves with bottom-up traversals such as those used in [12]. Associated with any traversal order is a characteristic function. This function specifies the position to be inspected after having inspected a prefix u. For example, a left-to-right traversal has the following characteristic function. Having visited a prefix u, this function chooses the leftmost position in $\mathcal{F}_u$ as the next position to visit. Note that if the prefix visited is simply a variable (i.e., no symbols have been inspected yet), then all traversal orders will specify the root position as the next one to inspect. We distinguish fixed and adaptive traversals using the following definition.

Definition 5 (Adaptive and Fixed Traversals) *An* adaptive *traversal is a top-down traversal wherein the position $p \in \mathcal{F}_u$ next visited is a function of the prefix u and the set $\mathcal{L}_u$. In a fixed traversal order p is simply a function of $\mathcal{F}_u$.*

By choosing the next position as a function of $\mathcal{L}_u$, an adaptive traversal can *adapt* itself to the given set of patterns. In contrast, a fixed-order traversal always makes a fixed choice from the positions in $\mathcal{F}_u$. For instance, a left-to-right traversal would always pick the left-most position in $\mathcal{F}_u$. A breadth-first traversal would pick the left-most position among positions of least length (i.e., the length of the integer string representing a position) from those in $\mathcal{F}_u$. Having defined the notion of adaptive traversals, we now proceed to present a generic algorithm for building adaptive automata.

2.1 Generic Algorithm to build Adaptive Automata

Figure 2 shows our algorithm *Build* for constructing an adaptive automaton. A state v of the automaton remembers the prefix u inspected in reaching v from the start state. Suppose that p is the next position inspected from v. Then there are transitions from v on each distinct symbol c that appears at p for any $l \in \mathcal{L}_u$. There will also be a transition from v on $\neq$ which will be taken on inspecting a symbol different from those on the other edges leaving v. The symbol $\neq$ appearing at a position p denotes the inspection of a symbol in the input that does not occur at

Procedure $Build(v, u)$

1. Let $\mathcal{M}$ denote patterns that match u.
2. If $\mathcal{M} \neq \phi$ and $\forall\, l \in \mathcal{L}_u \;\; \exists l' \in \mathcal{M}$ such that $priority(l') \geq priority(l)$ then
3. $\quad match[v] := \mathcal{M}$. /* State v announces a match for patterns in $\mathcal{M}$ */
4. else
5. $\quad p = select(u)$ /* *select* is a function to choose the next position to inspect */
6. $\quad pos[v] = p$ /* Next position to inspect is recorded in the *pos* field */
7. $\quad$ for each symbol for which $\exists l \in \mathcal{L}_u$ with $root(l/p) = c$ (for some nonvariable c) do
8. $\quad\quad$ create a new node v_c and an edge from v to v_c labeled c
9. $\quad\quad Build(v_c, u[p \leftarrow c(y_1, ..., y_{rank(c)})])$
10. $\quad$ If $\exists l \in \mathcal{L}_u$ with a variable at p or at an ancestor of p then
11. $\quad\quad$ create a new node $v_{\neq}$ and an edge from v to $v_{\neq}$ labeled $\neq$
12. $\quad\quad Build(v_{\neq}, u[p \leftarrow \neq])$

Figure 2: Algorithm for constructing adaptive automaton.

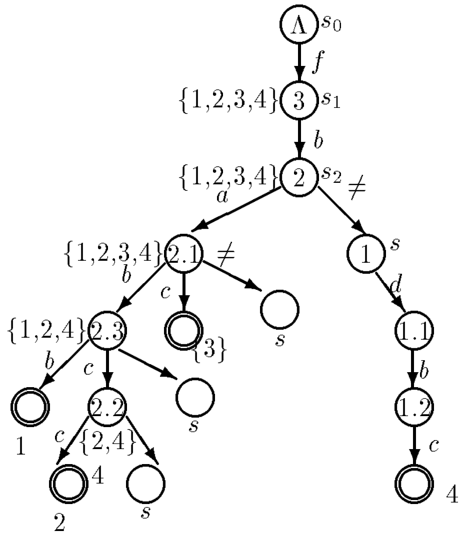
p in any pattern in $\mathcal{L}_u$. This implies that if a prefix u has a $\neq$ at a position p then every pattern that could potentially match an instance of u must have a variable at or above p.

There is an important distinction between *Build* and previously known algorithms (such as that of Christian [4] and Huet-Lévy[14]). *Build* is non-deterministic in that it does not specify a selection function to specify the next position to visit. This non-determinism enables us to reason about automaton construction *without any reference* to the traversal order used.

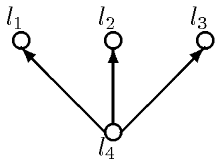
Procedure *Build* is recursive, and the automaton is constructed by invoking $Build(s_0, x)$ where s_0 is the start state of the automaton. *Build* takes two parameters: v, a state of the automaton and u, the prefix examined in reaching v. The invocation $Build(v, u)$ constructs the subautomaton rooted at v. In line 2, the termination conditions are checked. By definition of pattern match, we need to rule out possible matches with higher priority patterns before declaring a match for a lower priority pattern. Since the match set $\mathcal{L}_u$ contains all patterns that could possibly match the prefix u, we simply need to check that each pattern in the match set is either already in $\mathcal{M}$ or has a lower priority than some pattern in $\mathcal{M}$.

If the termination conditions are not satisfied then the automaton construction is continued in lines 5 through 12. At line 5, the next position p to be inspected is selected and this information is recorded in the current state in line 6. Lines 7,8 and 9 create transitions based on each symbol that could appear at p for any pattern in $\mathcal{L}_u$. In line 9, *Build* is recursively invoked with the prefix extended to include the symbols seen on the transitions created in line 8. If there is a pattern in $\mathcal{L}_u$ with a variable at or above p then a transition on $\neq$ is created at line 11 and *Build* is recursively invoked at line 12. The recursive calls initiated at lines 9 and 12 together will complete the construction of the subautomaton rooted at state v. Steps 10–12 will be skipped in case of typed systems if we have created transitions at step 9 corresponding to every symbol that can appear at the position p.

To illustrate the algorithm, consider the set of patterns shown in Figure 3. Pattern l_4 has a lower priority than any of the other patterns. There is no relationship among the priorities



$l_1 : f(\_, a(b, \_, b), b)$
$l_2 : f(\_, a(b, c, c), b)$
$l_3 : f(\_, a(c, \_, \_), b)$
$l_4 : f(d(b, c), \_, b)$



1. States are labeled by the next position to be inspected upon reaching that state.
2. All of the states marked with an s actually denote a subautomaton that is identical to the automaton rooted at the state labeled with 1.

Figure 3: Adaptive automaton constructed by *Build*.

of l_1, l_2 and l_3. The automaton construction begins with the invocation $Build(s_0, x)$. Observe that none of the patterns match the prefix x. Since the only fringe position in prefix x is Λ, $pos[s_0]$ is set as Λ. The state s_1 is created at line 8 and the edge between s_0 and s_1 is labeled by symbol f. Following this, a recursive call $Build(s_1, f(x, y, z))$ is made at line 9. At line 2 of this recursive call, we once again find that no pattern matches $f(x, y, z)$. Now assume that select at line 5 returns 3. Note that all patterns have the same nonvariable symbol at position 3 and so there is only one iteration of the loop at line 7. This results in invocation of $Build(s_2, f(x, y, b))$. Continuing this process, we obtain the automaton shown in Figure 3. The automaton has 25 states and each unlabeled state marked by s represents a subtree identical to that rooted at state labeled by 1.

Procedure *Build* as given is inefficient as it computes the sets $\mathcal{L}_u$ and $\mathcal{M}$ in each recursive call. Moreover it manipulates positions, which are *strings* of integers. The first source of inefficiency can be overcome easily by computing the match sets incrementally. To overcome the second source of inefficiency, positions can be encoded as integers in the range [1,S], where S is the sum of sizes of all patterns. A similar, but indirect technique is used in the context of compilation of pattern-matching for functional programming languages [2, 22, 19].

3 Synthesizing Traversal Orders

Observe that *Build* does not specify the selection function. In this section we present several techniques that can be used to construct the selection functions. These techniques can be either used independently or combined together as appropriate.

3.1 Measuring Size and Matching Time

The primary objective of a selection function is to reduce the automaton size and/or the matching time. Therefore it is important to know how we measure these quantities. A natural measure of the size of an automaton is the number of states in it. However, this measure has the drawback that minimization of total number of states is NP-complete, even for the simple case of patterns with no variables [5]. This makes it impossible to develop efficient algorithms that build an automaton of smallest size, unless $P = NP$. Even so, we would still like to show that certain algorithms are always better than others for reducing the size. The usual way to do this is to choose an alternative measure of size that is closely related to the original size measure, yet does not have the drawback of NP-completeness of its minimization. A natural choice in this case is the breadth of the automaton, which is related to the total number of states by at most a linear factor. (In practice, however, the factor is typically much smaller: note that if every node in the automaton has at least two children, then then the breadth is at least half as much as the number of states.)

As for matching time, it is easy to define the time to match a given term using a given automaton: it is simply the length of the path in the automaton from the root to the final state that accepts the given term. However, what we would like is a time measure that does not refer to input terms. We could associate an average matching time with an automaton, but this would require information that is not easily obtained: the relative frequencies with which each of the paths in the automaton are taken[3]. Therefore, instead of defining a time measure that totally orders the automata for a specific distribution of input terms, we use the following measure that partially orders them *independent* of the distribution. Let $MT(s, A)$ denote the length of the path in (automaton) A from the start state to the accepting state of the *ground* term s. (A ground term is a term that contains no variables.) If s is not accepted by A then $MT(s, A)$ is undefined.

Definition 6 *Let A and A' be two matching automata for $\mathcal{L}$, and t be any term. We say that $A \preceq_t A'$, meaning that A is more efficient than A' for matching instances of t, iff for every ground instance s of t, $MT(s, A) \leq MT(s, A')$.*

$A \preceq A'$ is a shorthand for $A \preceq_x A'$, where x is a variable. Note that $A \preceq A'$ means that A is more efficient than A' for matching *every* ground term.

When we need to get an idea of the work involved in matching various classes of patterns, it is useful to associate a quantitative measure of matching time with automata. We do this by first identifying the best possible automaton for the given set of patterns (as given by the partial order $\preceq$) and computing the average root-to-leaf path length of this automaton. When multiple minimal automata with different average root-to-leaf path lengths are possible, we will refrain from giving a quantitative measure.

3.2 Representative Sets

In this section, we introduce the important concept of a *representative set*. This notion makes use of the priorities among the patterns to eliminate those patterns from the match set for which no matches can be found. This happens for a prefix u whenever a match for a lower priority pattern implies a match for a higher priority pattern. For instance, consider the patterns in Figure 3 and the prefix $u = f(\_, a(b, \_, b), \_)$. Although $\mathcal{L}_u = \{l_1, l_4\}$, observe that a match for

[3]It is possible to assume that all terms over Σ are equally likely and derive a matching time on this basis, but such assumptions are seldom justified or useful in practice.

l_4 can be declared only if the 3[rd] argument of f is b. In such a case we declare a match for the higher priority pattern l_1. Inspecting any position only on behalf of a pattern such as l_4 is wasteful, e.g., inspection of position 1 for u is useless since it is irrelevant for declaring a match for l_1. We can avoid inspecting such positions by considering the representative set instead of a match set for a prefix u. A representative set is defined formally as follows:

Definition 7 *A representative set $\overline{\mathcal{L}_u}$ of a prefix u with respect to a set of patterns $\mathcal{L}$ is a minimal subset $\mathcal{S}$ such that the following condition holds for every l in $\mathcal{L}$.*

$$\forall t \geq u \;\; (l \text{ matches } t) \Rightarrow \;\exists l' \in \mathcal{S} \;\; [(l' \text{ matches } t) \wedge (priority(l') \geq priority(l))] \tag{1}$$

Using the definition of pattern match and through simplification, we can derive the following simpler condition equivalent to (1). Herein, we use the notation $P(l)$ to denote the priority of pattern l.

$$\forall t \geq (l \sqcup u)\; \exists l' \in \mathcal{S} \quad [(P(l') > P(l)) \wedge l' \uparrow t] \vee [(P(l') = P(l)) \wedge l' \leq t] \tag{2}$$

This condition captures our intuition about l that for any instance of u, either l does not match the instance (first part of the disjunction) or the match can be "covered" by another pattern in $\mathcal{S}$ (second part). Hence we refer to the property given by this condition as a *cover* property, and any set $\mathcal{S}$ satisfying the property as a *cover* for $\mathcal{L}$. A representative set is simply a minimal cover. We make the following observation about the transitivity of the cover property: **Observation 1:** If $\mathcal{S}_1$ is a cover for $\mathcal{L}$ and $\mathcal{S}_2$ is a cover for $\mathcal{S}_1$ then $\mathcal{S}_2$ is a cover for $\mathcal{L}$.

Note that if the set $\mathcal{L}$ contains multiple patterns with equal priority, then there may be a choice as to which of these patterns are retained in a representative set. Thus the definition does not always yield a unique representative set. Hence future references to $\overline{\mathcal{L}_u}$ or *representative set* refer to any one set that satisfies the above definition.

Laville's notion of accessible patterns [18] is similar to our notion of representative sets, but is not a minimal set. Our contributions here are that the minimality enables us to develop more efficient algorithms, and secondly, that we provide an algorithm for computing this set. The definition of accessible patterns in [18] does not yield such an algorithm and so it uses the notion of compatible patterns (which corresponds to our match set $\mathcal{L}_u$) in place of accessible patterns[4].

In the following section, we show how to design selection strategies using representative sets. The algorithmic aspects of computing the representative sets will be discussed in section 4.

3.3 Greedy Strategies

In this section we present many strategies for implementing the function *select* at a node v of the automaton. All these strategies select the next position based on *local information* such as the prefix and the representative set associated with v or its children. Let p denote the next position to be selected.

[4]We can also use $\mathcal{L}_u$ in place of $\overline{\mathcal{L}_u}$ in all the optimizations mentioned in this paper, but doing so may make the optimizations less effective. For instance, our algorithm for directly building optimal dag automata in section 6 will fail to identify some equivalent states if $\mathcal{L}_u$ is used in place of $\overline{\mathcal{L}_u}$.

1. Select a p such that the number of distinct nonvariables at p, taken over all patterns in $\overline{\mathcal{L}_u}$ is *minimized.* This strategy attempts to minimize the size by local minimization of breadth of the automata. It does not attempt to reduce matching time.

2. Select a p such that the number of distinct nonvariables at p, taken over all patterns in $\overline{\mathcal{L}_u}$ is *maximized.* The rationale here is that by maximizing the breadth, a greater degree of discrimination is achieved. If we can quickly distinguish among the patterns, then the (potentially) exponential blow-up can be contained. Furthermore, once we distinguish one pattern from the others, we no longer inspect unnecessary symbols and so matching time can also be improved.

3. Select a p such that the number of patterns having nonvariables at p is maximized. The motivation for this strategy is that only patterns with variables at p are duplicated in the representative sets of the descendents of the current state v of the automaton. By minimizing this number of patterns that are duplicated, we can contain the blow-up. Furthermore, this choice minimizes the probability of inspecting an unnecessary position: it is a necessary position for the most number of patterns.

4. Let $\overline{\mathcal{L}_1}, ..., \overline{\mathcal{L}_r}$ be the representative sets of the children of v. Select a p such that $\Sigma_{i=1}^{r}|\overline{\mathcal{L}_i}|$ is minimized. Note that the main reason for exponential blow-up is that many patterns get duplicated among the representative sets of the children of v. This strategy locally minimizes such duplication (since $\Sigma_{i=1}^{r}|\overline{\mathcal{L}_i}|$ is given by the size of $\overline{\mathcal{L}_u}$ plus the number of number of patterns that are duplicated among the representative sets of the children states.) For improving time, this strategy again locally minimizes the number of patterns for which an unnecessary symbol is examined at each of the children of v.

All of the above greedy strategies suffer from the drawback that:

Theorem 8 *For each of the above strategies there exist pattern sets for which automata of smaller size can be obtained by making a choice different from that given by the strategy.*

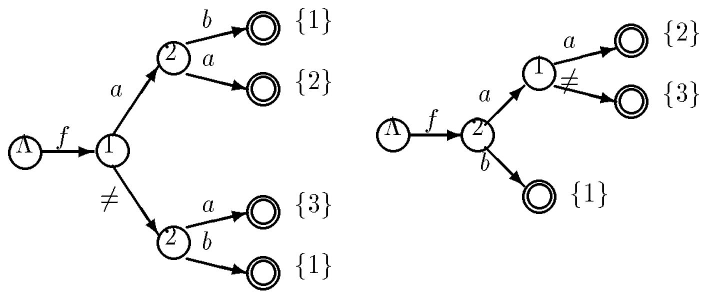
Proof: It is quite straight forward to give an example for strategy 1 and we omit it here. For strategies 2, 3 and 4 consider the following set of patterns with equal priorities:

$$\begin{array}{lll} f(a, a, \_, \_) & f(b, \_, a, \_) & f(c, \_, \_, a) \\ f(\_, b, b, b) & f(\_, c, c, c) & f(\_, d, d, d) \end{array}$$

After inspecting the root, all these strategies will choose one of positions 2, 3 or 4. It can be shown (by enumerating all possible matching automata for these patterns) that the smallest breadth and number of states obtainable by this choice are 20 and 49 respectively. These figures can be reduced to 15 and 45 respectively by choosing position 1. ■

The construction of this example is quite intricate. The key idea is to make each of the pattern sets $\{1, 4, 5, 6\}$, $\{2, 4, 5, 6\}$ and $\{3, 4, 5, 6\}$ strongly-sequential[5] whereas any set containing two of the first three patterns and one of the last three is not. Such a choice ensures that if a traversal order first inspects position 1 after the root position, then every subautomata below this state

[5]In strongly-sequential systems, any prefix u with $|\mathcal{L}_u| \geq 1$ must have an index.



Priority	Pattern
1	$f(\_, b)$
2	$f(a, a)$
3	$f(\_, a)$

Figure 4: Illustration of size and matching time reduction due to interchange step.

will match a set of strongly sequential patterns. Since optimal automata can be constructed for such patterns, these subautomata can be of a small size. In contrast, any other choice of position to inspect will result in match sets that are not strongly sequential, and hence lead to subautomata that are larger in size.

The contrived nature of the example shows that although it is possible for these strategies to fail, such failures may be rare. Even when they fail, as in the above example, they still appear to be significantly better than fixed order traversals. For instance, right-to-left traversal constructs an automaton with breadth 30 and number of states 60. Left-to-right traversal reduces these figures to 24 and 58, which is still more than that obtained by using any of the strategies 2,3 and 4.

Conceptually, the problems outlined above with regards to choice of next inspected position arise in the context of minimizing matching time as well. However, since the definition of $\preceq$ does not permit us to compare arbitrary automata, such a result cannot be established without making assumption about the distribution of input terms[6].

3.4 Selecting Indices

We now propose another important local strategy that does not suffer from the drawbacks of the greedy strategies discussed in the previous section. The key idea is to inspect the index positions in u whenever they exist. We show that this strategy yields automata of smaller (or same) size and superior (or same) matching time than that obtainable by any other choice. The importance of index was known only in the context of strongly-sequential systems. Our result demonstrates its applicability to patterns that are not strongly-sequential also.

Consider the two automata shown in Figure 4. The second automaton is obtained from the first by interchanging the order in which positions 1 and 2 are visited. Observe that by inspecting the index position 2 before the non-index position 1, the second automaton improves the space requirements. It also requires less time to identify a match since each path in the second automaton examines only a subset of the positions examined in the corresponding path(s) in the first automaton. The following theorem formalizes the interchange operation and outlines its benefits.

[6]One must not, however, conclude our choice of $\preceq$ is inappropriate for comparing matching times. As mentioned before, it is essentially the only way to compare matching time when we have no knowledge of the distribution of input terms.

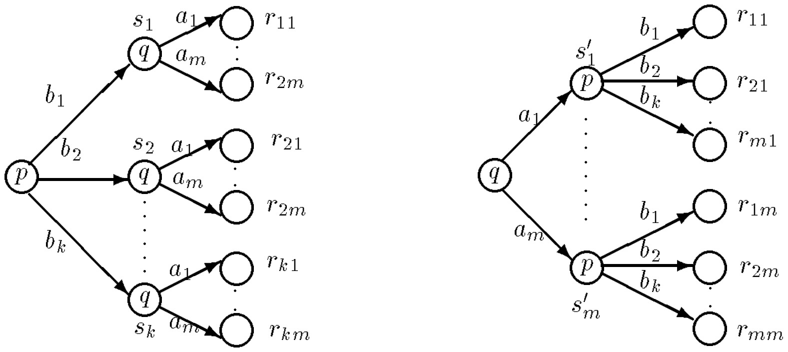


Figure 5: Automaton before and after an interchange step.

Theorem 9 *Let v be a state in an automaton B, the prefix corresponding to v be u and $pos[v] = p$. If q is an index of u such that $pos[w] = q$ for every child w of v then we can obtain an automaton B' from B by interchanging the order of inspection of p and q in such a way that $|B'| \leq |B|$ and $B' \preceq B$.*

Proof: First we describe the construction of B' from B. Let A be the subautomaton of B that is rooted at v and u be the prefix that has been inspected in reaching v. The construction of B' takes place in two steps.

Step 1: Interchange the order of inspection of positions p and q, without changing any other part of the automaton, as shown in Figure 5. In this figure $a_1, \ldots, a_m$ are all the symbols that appear at q in any pattern in $\overline{\mathcal{L}_u}$. Similarly, $b_1, ..., b_k$ are all the symbols that appear at p for any pattern in $\overline{\mathcal{L}_u}$. Note b_k is $\neq$ if some pattern in $\overline{\mathcal{L}_u}$ has a variable at or above q. It is also possible for a_m to be $\neq$. Also note that some of the states r_{ij} may not be present because no pattern in $\mathcal{L}_u$ has a b_i at p and a_j at q. Such r_{ij}'s denote empty subautomata. Note that the prefix inspected at r_{ij} is the same in the automaton before and after the interchange. Hence the structure of the subautomata rooted at an r_{ij} are also identical before and after the interchange. Consequently, it is possible to perform the above interchange in the order of inspection of p and q.

Step 2: Replace each s'_i that satisfies the following condition by r_{i1}.

$$\forall l \in \overline{\mathcal{L}_{prefix(s'_i)}} \;\; l/p \text{ is a variable}$$

where the notation $prefix(s'_i)$ is used to denote the prefix inspected on reaching the state s'_i of the automaton. States such as s'_i that satisfy the above condition can exist whenever p is *not* an index. When this condition is satisfied, all the subautomata rooted at $r_{1i}, r_{2i}, \ldots, r_{mi}$ that are below s'_i will be identical and the position p need not be inspected at all. Therefore, in the second step, we replace such s'_i by r_{1i}.

This completes the construction of B'. We now show that $|B'| \leq |B|$. This is easy to see. Step 1 does not change the breadth of the automaton (which is our measure of size) and Step 2 can only reduce the breadth.

To show that $B' \preceq B$, we proceed as follows. The construction above (steps 1 and 2) induces a mapping $\mathcal{I}$ from the final states in B to final states in B' (which are a subset of final states in B). For a final state v_1 in B, its image $v_1' = \mathcal{I}(v_1)$ in B' is defined as follows:

Case 1: v_1 is a descendent of an r_{ij} such that the state s_i' was eliminated (by replacing it with r_{i1}) in step 2 of the construction: By remarks made in step 2, r_{ij} are identical for $1 \leq j \leq m$. This identity defines a natural correspondence between final states in an r_{ij} and final states in r_{i1}. We take $\mathcal{I}(v_1)$ in this case to be the corresponding state within r_{i1}.

Case 2: Otherwise, (i.e., not case 1), v_1 appears unchanged in B' and so we take $\mathcal{I}(v_1) = v_1$.

Now, consider any term s for which a match is announced by B and let v_1 be the final state reached. It is easy to see that the state $\mathcal{I}(v_1)$ will be reached when B' is used for matching s. Moreover, the positions examined on the path to $\mathcal{I}(v_1)$ is a subset of those inspected in reaching v_1. By definition of $\preceq$, this implies that $B' \preceq B$. ∎

Observe that the above theorem can be used only if all children of v inspect the same index position q. However even if some children of v inspect position q' other than q, still it is possible to globally rearrange the automaton to inspect q at v, and thus achieve the benefits of inspecting the index position early. This is because q is an index position and hence must be inspected before declaring a match. Therefore it will appear in each path from v to any accepting state in the subautomaton rooted at v.

Theorem 10 *Let v be a state in an automaton B, u be the prefix corresponding to v and $pos[v] = p$. If q is an index of u then we can obtain another matching automaton B' from B by replacing its subautomaton A rooted at v by another (sub)automaton A' such that $|B'| \leq |B|$ and $B' \preceq B$.*

Proof: We construct B' by repeating the interchange operation. The proof that B' can be so obtained is by straight forward induction on the height of the subautomaton A. ∎

Although the theorem only asserts that size and matching time of A' is no larger than that of A, Figure 4 shows that they can be strictly smaller for A'. We remark that by repeating the interchange steps as above, size can sometimes be reduced by as much as an exponential factor and time by $O(n)$, where n is the number of patterns.

We point out that the interchange operations mentioned above constitute merely a proof technique; they play no part at all in the actual construction of the automaton. In the actual construction, the same effect is obtained simply by modifying the selection function to inspect indices whenever possible.

3.5 Adaptive Traversal Orders for Lazy Functional Languages

In lazy functional languages, evaluation (of input terms) is closely coupled with pattern-matching. Specifically, a subterm of the input term is evaluated only when its root symbol needs to be inspected by the pattern-matcher. If there are subterms whose evaluation does not terminate, then an evaluator that uses an algorithm that identifies matches without inspection of such subterms can terminate, whereas use of algorithms that do inspect such subterms will lead to non-termination. Since the set of positions inspected to identify a match is dependent on the traversal order used, the termination properties also depend upon the traversal order. In order to make sure that the program terminates on input terms of interest to the programmer, the programmer (sometimes) has to reason about the traversal order used. In particular, the programmer can code his/her program in such a way that (for terms of interest to him/her) the pattern-matcher will inspect only those subterms whose evaluation will terminate. This implies that the programmer must be made aware of the traversal order used *even before the program is written* — thereby ruling out synthesis of *arbitrary* traversal orders at compile time. Given this constraint on preserving termination properties, a natural question is whether the traversal order can be "internally changed" by the compiler in a manner that is transparent to the programmer. Specifically, given a traversal order T that is assumed by the programmer, our goal is to synthesize a new traversal order $S(T)$ such that any evaluation algorithm that terminates with T will also do so with $S(T)$. We devise such a traversal order in this section.

3.5.1 Preliminaries

We first tighten our original definition of a traversal so as to capture the important aspect of determinacy in the order in which various positions visited.

Condition 1 (Monotonicity) Suppose that a traversal order T selects p as the next position to be visited for a prefix u. T is said to be monotonic iff for any prefix $u' \geq u$, the following condition holds: If u'/p is a variable and l/p is a nonvariable in some $l \in \overline{\mathcal{L}_{u'}}$ then the traversal once again selects p.

Monotonic traversals include most known traversal orders such as depth-first and breadth-first, as well as variations of these without left-to-right bias. An example of a traversal that is not monotonic is given by the following select function:

$$
\begin{aligned}
select(f(x,y,z)) &= 1 && (3)\\
select(f(x,a,z)) &= 3 && (4)
\end{aligned}
$$

We also require that the selection function make its decision only based portions of the prefix that are "relevant," as given below.

Condition 2 Let u_1 and u_2 be two prefixes with identical representative sets. Suppose that the prefixes differ only in subterms appearing at positions where every pattern in this representative set has a variable. Then the selection function must choose the same position to inspect for u_1 and u_2.

Clearly, the symbols appearing at such positions are irrelevant for determining a match – and consequently irrelevant for selecting which positions to inspect next. Therefore we require that the selection function choose the same position to inspect for u_1 and u_2 in such a case. Henceforth, we consider only selection functions that satisfy the above two conditions.

Given a monotonic traversal T, we define $S(T)$ as follows:

Definition 11 *$S(T)$ is any traversal order characterized by the following selection function for any prefix u:*

- *If u has indices then arbitrarily choose one of them.*
- *Otherwise, choose the position that would be selected by T.*

3.5.2 Size and Matching Time Improvement using $S(T)$

Let B_0 be a matching automaton that uses traversal order T. We will now show that any automaton B' using $S(T)$ can be constructed from B_0 through the interchange operations of Theorem 9 and 10. It then follows from Theorem 10 that $S(T)$ improves space and matching time requirements over an automaton using T. Also note that (as established in proof of Theorem 9) after the interchange step, each path in the new automaton examines a subset of the positio ns examined in the corresponding path in the old automaton. Therefore, a lazy evaluation algorithm based on the new automaton will terminate in every case when the algorithm terminated with the old automaton.

For the proof that the interchange operations lead to an automaton using $S(T)$, we need to go back to the construction in Figure 5. Recall that v is a state in B corresponding to a prefix u with $pos[v] = p$, and for every child w of v, $pos[w]$ is an index position q. Let B' be the automaton obtained by interchanging the order of inspection of p and q. To relate the traversal orders used in B and B', we define the following correspondence mapping $\mathcal{C}$ from each state v' of B' that inspects a non-index position to a set of states in B.

Case 0: $v' = v$: Since v inspects an index in B', $\mathcal{C}(v)$ need not be defined.

Case 1: either v' is not a descendent of v, or v' is a descendent of a state r_{ij} such that the state s'_i was not eliminated in step 2: It is clear that in this case, the state appears unchanged in B and so $\mathcal{C}(v') = \{v'\}$.

Case 2: $v' = s'_i$ for some i such that s'_i was *not* eliminated in step 2: Then $\mathcal{C}(v') = \{v\}$.

Case 3: v' is a descendent of an r_{i1} such that s'_i was eliminated by step 2 of the construction: Since $r_{i1}, ..., r_{im}$ are identical in this case, there is a natural one-to-one correspondence between states in r_{ij} and those in r_{i1}. $\mathcal{C}(v')$ in this case will be the set of all the states in $r_{i1}, ..., r_{im}$ that correspond to v' in this manner.

We make the following observations about the mapping $\mathcal{C}$.

Observation 12 *For every v' in B' that examines a non-index position*

$$\forall s \in \mathcal{C}(v') \;\; pos[v'] = pos[s]$$

Although this observation shows that the positions inspected in corresponding states are identical, it does not imply any thing about the traversal orders used, since no assertion is made about the prefixes inspected at these states. To establish a relationship between the selection functions used in B and B', we show

Lemma 13 *Let B' be an automaton obtained by performing one interchange step on B. Let v' be any state in B' that examines a non-index position. Also let sel be the selection function used in B in any state that does not choose an index to inspect next, i.e., for any state s in B with $pos[s]$ not an index of $prefix(s)$, $sel(prefix(s)) = pos[s]$. Then*

$$pos[v'] = sel(prefix(v'))$$

Proof: The proof is by analysis of each of the cases defining $\mathcal{C}$. In case 1, the prefix inspected at v' is identical to that inspected at the (only) state s in $\mathcal{C}(v')$. Thus $pos[v'] = pos[s] = sel(prefix(s)) = sel(prefix(v'))$. In case 2, it is easy to see that for the only element v in $\mathcal{C}(v')$, $prefix(v') = prefix(v)[q \leftarrow a_i(x_1, ...x_{rank(a_i)})]$ for some symbol a_i. Also note that $prefix(v')/p$ is a variable since p has not been inspected in reaching v'. In addition, it is not possible for every pattern in $\overline{\mathcal{L}_{prefix(v')}}$ to have a variable at or above p – in such a case, the state v' would have been eliminated by step 2 of our construction. These facts, together with monotonicity, imply that $p = sel(prefix(v')) = sel(prefix(v)) = pos[v] = pos[v']$. In case 3, let s be any state in $\mathcal{C}(v')$. It is easy to see (by step 2 of construction) that $prefix(s)$ is identical to $prefix(v')$ in all positions except p. Moreover, by criteria for applying step 2 of construction of B', every pattern $l \in \overline{\mathcal{L}_{prefix(v')}}$ has a variable at or above p. Therefore the symbol at p is irrelevant for pattern matching. By condition 2 on selection function, $sel(prefix(v'))$ must be the same as $sel(prefix(s)) = pos[s] = pos[v']$. ■

We now extend the lemma 13 so that it holds for arbitrary number of interchange steps between B and B'.

Lemma 14 *Let B' be an automaton obtained by performing zero or more interchange steps on B_0. Let v' be any state in B' that examines a non-index position. Also let sel be the selection function used in B_0 in any state that does not choose an index to inspect next, i.e., for any state v_0 in B_0 with $pos[v_0]$ not an index of $prefix(v_0)$, $sel(prefix(v_0)) = pos[v_0]$. Then*

$$pos[v'] = sel(prefix(v'))$$

Proof: By simple induction on the number of interchange steps used to obtain B' from B_0. ■

Lemma 15 *Let T be a monotonic traversal order. Any automaton using $S(T)$ can be obtained by performing the interchange operations on an automaton that uses T.*

Proof: Let B_0 be an automaton that uses traversal order T. Use the construction outlined in Lemmas 13 and 14 repeatedly on the automaton B_0 to obtain another automaton B' that examines indices as early as possible. In this automaton B' (see proof of Lemma 14), if a state v' in B' does not inspect an index position then it inspects the position specified by selection function used in B_0, i.e., a position given by the traversal order T. By Definition 11, this means that B' uses $S(T)$, so we need only show that the above construction can be used to obtain an automaton for *any* traversal order that follows Definition 11. Note that, given a T, $S(T)$ is uniquely determined, except for the order in which index positions are inspected. Clearly, all possible permutations of such positions can be obtained using the interchange operation. Thus, any automaton that follows Definition 11 can be obtained through interchange operations from B_0. ■

Theorem 16 *Let T be a monotonic traversal. Size and matching time can never become worse if $S(T)$ is used in place of T. Moreover, each path in the automaton using $S(T)$ examines a subset of the positions examined on the corresponding path in T.*

Proof: Since the interchange operations can only improve space and matching time, the first part of the theorem is immediate. For the second part, note that, by construction of the interchange operation, the positions inspected on a root-to-leaf path in the automaton after the interchange is a subset of those positions inspected before the interchange. ∎

We remark that the subset property ensures that any evaluation algorithm that terminates with traversal order T will terminate if $S(T)$ is used in place of T.

4 Computational Aspects

In order to implement the selection function described in the previous section we must develop an algorithm to compute the representative set. Similarly we must have a method to identify indices of a prefix to incorporate space and matching time optimizations. In the following we discuss the algorithmic aspects of these problems.

4.1 Computing Representative Sets

We now present an efficient procedure for computing representative sets in untyped systems.

Procedure $computeRepSet(u, \mathcal{L}_u)$

1. $\mathcal{L}' := \mathcal{L}_u$
2. While $\exists l_1, l_2 \in \mathcal{L}'\ \ [(l_1 \neq l_2) \wedge (l_1 \sqcup u \geq l_2) \wedge (priority(l_2) \geq priority(l_1))]$ do
3. delete l_1 from $\mathcal{L}'$

Theorem 17 *Procedure computeRepSet computes the representative set of u in $O(nS)$ time for untyped systems, where n is the number of patterns in $\mathcal{L}_u$ and S the sum of sizes of these patterns.*

Proof: The time complexity result can be readily established, so we focus only on correctness. We first establish that $\mathcal{L}'$ is a cover by inducting on the number of times the loop at lines 2–3 is executed. In the base case $\mathcal{L}' = \mathcal{L}_u$ is obviously a cover. For the induction step, let $\mathcal{L}_1$ and $\mathcal{L}_2$ denote the values of $\mathcal{L}'$ before and after the deletion of l. By induction hypothesis and transitivity of the cover property, we need only show that $\mathcal{L}_2$ is a cover for $\mathcal{L}_1$. Given the condition at line 2 of the algorithm, it is easy to see that the cover condition (2) on page 11 holds with $l = l_1, l' = l_2, \mathcal{L} = \mathcal{L}_1$ and $\mathcal{S} = \mathcal{L}_2$.

We now prove by contradiction that the set $\mathcal{L}'$ returned by *computeRepSet* is a minimal cover. Assume that it is not, so there must be an $l \in \mathcal{L}'$ that can be deleted without affecting the cover property. This assumption means that for every $t \geq l \sqcup u$, the body of condition (2) holds with $l' \neq l$. Now consider the term t obtained by instantiating all variables in $l \sqcup u$ by $\neq$. By construction of t, if any pattern l' unifies with t it must be the case that $t \geq l'$, and $l \sqcup u \geq l'$. Thus the body of condition (2) implies that $\exists l' \in \mathcal{L}'\ [(priority(l') \geq priority(l)) \wedge (l' \leq l \sqcup u)]$. This being the condition tested at line 2 of the algorithm, such an l would not have been present in $\mathcal{L}'$ – a contradiction. ∎

The proof of minimality hinges on the ability to obtain a term by instantiating variables with $\neq$. This is not always possible in a typed system since the term obtained by instantiating in this manner may violate the type discipline. Therefore, in typed systems, the above procedure computes a cover that is not necessarily minimal. In fact the following theorem shows that we are unlikely to have efficient procedures for computing a minimal cover (i.e., $\overline{\mathcal{L}_u}$) in typed systems.

Theorem 18 *Computing $\overline{\mathcal{L}_u}$ is NP–complete for typed systems.*

Proof: The problem of computing $\overline{\mathcal{L}_u}$, when posed as a decision problem, takes the form "Does $l \in \overline{\mathcal{L}_u}$?" By definition of representative set, this problem is equivalent to determining if there exists a term t (subject to the type discipline) such that the following condition holds:

$$(t \geq u) \wedge (t \geq l) \wedge (\forall l' \ \ (priority(l') > priority(l)) \Rightarrow t \not\uparrow l')$$

It is easy to see that this problem is in *NP* since we need only guess a term t and check that it conforms to the type discipline, and that it is an instance of u and l but does not unify with any l' of higher priority. All these can clearly be accomplished in polynomial time. To show that the problem is *NP*–complete, we will reduce the satisfiability problem (SAT [8]) to identifying such a t.

Let $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_n$ be an instance of satisfiability problem where φ_i is a disjunct of literals of the form z or $\neg z$, $z \in \{z_1, \ldots, z_m\}$. We transform this into an instance of determining whether a pattern belongs to $\overline{\mathcal{L}_u}$. Consider a (function) symbol f taking $m+1$ arguments and constrained by a type discipline such that each of these arguments must be either a or b. Assume that f is defined using the following $n+1$ patterns $l_1, ..., l_{n+1}$ with textual order priority, i.e., $priority(l_i) > priority(l_{i+1})$ for $1 \leq i \leq n$. l_{n+1} is of the form $f(a, y_1, ..., y_m)$. For $1 \leq i \leq n$, $l_i = f(y_0, s^i_1, ..., s^i_m)$, where s^i_j for $1 \leq j \leq m$ is given by

$$\begin{aligned} s^i_j &= a, && \text{if } z_i \text{ appears in } \varphi_i && (5)\\ s^i_j &= b, && \text{if } \neg z_i \text{ appears in } \varphi_i && (6)\\ s^i_j &= \_ && \text{otherwise.} && (7) \end{aligned}$$

Now consider the problem of checking if there is an instance of $u = f(x_0, ..., x_m)$ that does not unify with any of the first n patterns. (This will determine if the $(n+1)^{\text{th}}$ pattern belongs to the representative set for the prefix $f(x_0, ..., x_m)$.) Suppose that there is such an instance $t = u\sigma$. Consider the following truth assignment $\mathcal{T}$ derived from t:

$$\begin{aligned} z_j &= false, && \text{if } \sigma(x_j) = a && (8)\\ z_j &= true, && \text{otherwise, i.e., } \sigma(x_j) = b \text{ or } \sigma(x_j) = \_. && (9) \end{aligned}$$

Since $t = u\sigma$ does not unify with l_i for $1 \leq i \leq n$, we know that there exists some s^i_j such that either $s^i_j = a \neq \sigma(x_j) = b$, or $s^i_j = b \neq \sigma(x_j) = a$. In the former case, z_j appears in φ_i (by (5)) and z_j is set to $true$ (by (9)) and hence φ_i is satisfied. In the latter case, $\neg z_j$ appears in φ_i (by (6)) and z_j is set to $false$ (by (8)) and hence φ_i is once again satisfied. Since the above argument holds for $1 \leq i \leq n$, we have constructed a solution for SAT.

Now we show that whenever the input to SAT problem is satisfiable, $l_{n+1} \in \overline{\mathcal{L}_u}$. Suppose that $\mathcal{T}$ is a truth assignment that satisfies φ. Consider the following substitution σ such that $\sigma(x_0) = a$ and for other x_j, $1 \leq j \leq m$, σ is given by:

$$\begin{aligned} \sigma(x_j) &= b, && \text{if } z_j = true \text{ in } \mathcal{T} && (10)\\ \sigma(x_j) &= a, && \text{otherwise.} && (11) \end{aligned}$$

Since $\mathcal{T}$ satisfies φ_i for $1 \leq i \leq n$, it must be the case that either there is a z_j in φ_i that is set to *true* by $\mathcal{T}$, or there a $\neg z_j$ in φ_i such that z_j is set to *false* by $\mathcal{T}$. In the former case, it is easy to see that $s^i_j = a$ (by (5)) and $\sigma(x_j) = b$ by (10) and hence $t = u\sigma$ does not unify with l_i. In the latter case, $s^i_j = b$ and $\sigma(x_j) = a$ and so t once again does not unify with l_i. Moreover, since $\sigma(x_0) = a$, $t \geq l_{n+1}$ and thus we can answer "yes" to the question of whether $l_{n+1} \in \overline{\mathcal{L}_u}$. ■

4.2 Index Computation

Recall that an index for a prefix u is a position on its fringe that must be inspected to announce a match for any pattern in $\mathcal{L}_u$ (or $\overline{\mathcal{L}_u}$). In the absence of priorities, the indices of l are exactly those fringe positions wherein l has a nonvariable. With priorities, however, we may have to inspect positions wherein l has a variable in order to rule out a match for higher priority patterns. To identify these variable positions (that are indices) Laville [17] proposed an indirect method. In this method the prioritized patterns are first transformed into an equivalent set of unprioritized patterns, and then indices are identified using this set. Specifically, for each pattern l, the transformation generates a set $\mathcal{M}_l$ of its instances (called *minimally extended patterns*) that are not instances of any higher priority pattern. For typed systems, only those instances that observe the type discipline are generated. The transformed system is $\bigcup_{l \in \mathcal{L}} \mathcal{M}_l$. Now the indices w.r.t. the prioritized patterns are identical to those w.r.t. the unprioritized set $\bigcup_{l \in \mathcal{L}} \mathcal{M}_l$.

Puel and Suarez [23] developed a compact representation for the sets $\mathcal{M}_l$ based on the notion of *constrained terms*. A constrained term is of the form $\{t|\varphi\}$, where t is a term and φ is a constraint obtained by combining atomic constraints of the form $s_1 \neq s_2$ using conjunction and disjunction. The semantics of constrained terms is given by first regarding a term t with variables as denoting the set $\mathcal{I}(t)$ of its instances. Terms that satisfy the atomic constraint $s_1 \neq s_2$ must belong to the set $\mathcal{I}(s_1) - \mathcal{I}(s_2)$. A constraint $\varphi \vee \psi$ is satisfied by all (and only) terms that satisfy either φ or ψ. Similarly, the constraint $\varphi \wedge \psi$ is satisfied by all (and only) terms that satisfy φ as well as ψ. (We assume that a collection of atomic formulas used to construct a constraint do not share variables[7]). Henceforth we use several methods to simplify constrained terms. These methods are quite intuitive and their correctness readily follows from the above semantics. (For a more formal treatment of constrained terms, see [23].)

Using constrained terms the set $\mathcal{M}_l$ can be represented compactly as

$$\{l|(l \neq l_1) \wedge \cdots \wedge (l \neq l_k)\}$$

where $l_1, \ldots, l_k$ are all the patterns with priority greater than l. Since it is not apparent how indices can be computed when constraints are of this form, Puel and Suarez first transform this constrained term so that all the constraints are on variables in l. This yields a formula in conjunctive normal form (CNF), which is then converted to an equivalent constraint in disjunctive normal form (DNF). In the DNF form, indices can be easily picked: a variable x is an index for a pattern l if a constraint on x appears in every conjunction in the DNF. The above conversion of the constraint on a pattern l from CNF to DNF is very expensive and can take $O(|l|^n)$ time in the worst case. Therefore Puel and Suarez's algorithm, which is based on such a conversion, has exponential time complexity for *both typed and untyped systems.* Laville's algorithm is also

[7]This is to prevent having constraints such as $x \neq a(y) \wedge y \neq b$ (which shares a variable y between two constraints) that will complicate the development of the materials in the rest of the section.

exponential since the size of the set of minimally extended patterns can be exponential in the size of the original patterns. In contrast, we now present the first polynomial-time algorithm for untyped systems that operates directly on the original patterns.

4.3 Algorithm for Index Computation in Untyped Systems

The index for a prefix is computed in two steps. First we compute the set of indices of the prefix w.r.t. each of the constrained patterns in $\overline{\mathcal{L}_u}$ individually. The intersection of the sets thus computed yields the indices of the prefix w.r.t. $\overline{\mathcal{L}_u}$. We compute the indices of a prefix u w.r.t. to a single constrained pattern l as follows:

Let $l_1, l_2, \cdots, l_k$ be the patterns in $\overline{\mathcal{L}_u}$ that have priority over l and also unify with l. Following two steps specify the indices of u w.r.t. l.

1. Each variable position p in u such that l/p is a nonvariable.

2. Each variable position p in u such that l/p is a variable and p is the only position to be instantiated in $l \sqcup u$ to determine (or rule out) a match for some higher priority pattern l_j. More formally, there is a term s such that $(l \sqcup u)[p \leftarrow s] = l_j$.

We now illustrate how to use above two steps on the following patterns (with textual order priority) and the prefix $u = f(x, y, z)$.

$$\begin{aligned} l_1 &= f(a, b, c) \\ l_2 &= f(a, \_, \_) \\ l_3 &= f(\_, \_, c) \end{aligned}$$

Observe that x, y and z are all indices of l_1 by step 1. The only index for l_2 is x by step 1, since step 2 does not yield any additional indices. This is because $l_2 \sqcup u = f(a, y, z)$, and neither y nor z is the *only* variable whose instantiation can eliminate the match for l_1. Hence they do not satisfy the conditions in rule 2. For l_3, z is an index by step 1. In step 2, $l_3 \sqcup u = f(x, y, c)$, and comparing it with the higher priority pattern l_2, we find that x is the only variable that needs to be instantiated to rule out a match with l_2. Therefore x is an index by step 2. The intersection of all these positions is x which is therefore an index for u. Note that this method takes $O(nS)$ time to compute all indices.

We remark that a similar algorithm was suggested by Laville as a heuristic for fast index computation. However, the question of the power (or completeness) of the heuristic is not addressed at all. In contrast we show:

Theorem 19 *The algorithm for computing indices in untyped systems is sound and complete.*

Proof: Soundness of step 1 is obvious. For soundness of step 2, note that $(l \sqcup u)[p \leftarrow s] = l_j$, and also that (by definition of pattern match) $t \geq (l \sqcup u)$ and t does not unify with l_j. From these facts, it follows that t/p does not unify with s, which means that t/p is a nonvariable.

For completeness, we need to show that if a position p in u is not selected by steps 1 or 2 then it is not an index of $l \in \overline{\mathcal{L}_u}$. This is accomplished by giving a term t such that l matches

t, yet t/p is a variable. The term t is obtained by instantiating all the variables in $l \sqcup u$, except the variable at p, by the symbol $\neq$. Let $l_1, ..., l_k$ be all the patterns in $\overline{\mathcal{L}_u}$ with priority higher than that of l. We now show that l matches t. Since $t \geq l$, we need only show that t does not unify with any of $l_1, ..., l_k$. We prove this by contradiction. Assume that t unifies with l_j. Since $\neq$ is not present in any pattern, the only way t can unify with l_j is if l_j has variables at all the fringe positions of $l \sqcup u$ except possibly p. But l_j/p cannot be a variable as $l \in \overline{\mathcal{L}_u}$ (otherwise $(l \sqcup u) \geq l_j \Rightarrow l \notin \overline{\mathcal{L}_u}$ by the condition on line 2 of algorithm *computeRepSet*). Therefore l_j/p must be a nonvariable. Now note that l_j has a variable in all fringe positions of $l \sqcup u$ except p, thereby satisfying the requirement in step 2. This means that p would have been chosen as an index by step 2 of the above algorithm, but it is not – a contradiction. ■

4.4 Index Computation in Typed Systems

Unfortunately, computing indices for typed system is very hard. Although the method we developed for untyped system is sound, it is not complete for typed systems. In fact, it is very unlikely that we have polynomial time algorithms for computing indices in typed systems. The intuitive reason behind this complexity gap between typed and untyped systems can be explained by drawing the following analogy. Observe that, in the Puel and Suarez's approach, the literals in constraints generated are all of the form $x \neq t$ where x is a variable and t an arbitrary term. Such constrained terms are analogous to boolean formulas with only negative literals and hence are trivially satisfiable (by a truth assignment that assigns $false$ to every literal). Similarly index computation is simple in untyped systems. However, in typed systems, there are implicit positive constraints introduced by the type discipline. Thus we have a constraint that contains both positive and negative literals. Such a constrained term is analogous to a boolean formula with both positive and negative literals and satisfiability problem now becomes harder.

Theorem 20 *Index computation for typed systems is co-NP complete.*

Proof: The index selection problem, when posed as a decision problem, takes the form "Does u possess an index w.r.t. pattern set $\mathcal{L}$?" To show that this problem is in *co-NP*, we need to show that the problem of deciding whether u has no index is in NP. To do this, let $p_1, ..., p_r$ be the set of fringe nodes of u. We first guess r instances $t_1, ..., t_r$ of u such that t_i/p_i is a variable for $1 \leq i \leq r$. Then we verify that (at least) one pattern in $\mathcal{L}$ matches each t_i and if so we declare that u has no index. All this can clearly be accomplished in polynomial time and hence the problem of determining whether u does not possess an index is in NP and so the index selection problem is in *co-NP*.

To show that the problem is *co-NP*–complete, we reduce the complement of satisfiability to this problem. Let $\varphi_1 \wedge \cdots \wedge \varphi_n$ be an instance of satisfiability problem where φ_i is a disjunct of literals of the form x or $\neg x$, $x \in \{x_1, \ldots, x_m\}$. We transform this into an index computation problem as follows. Consider the following system consisting of $n+1$ patterns, with textual order priority. The root of each pattern is labeled by a $(m+1)$ arity function symbol f (which once again stands for the common prefix shared by all the patterns). The last m arguments of f are of a type that consists of nonvariables drawn from the set $\{a, b\}$. The $(n+1)^{\text{th}}$ pattern is of the form $f(x_0, x_1, ..., x_m)$. To specify the first n patterns, let $t_1, ..., t_n$ be terms that do not unify with each other. Also assume that there is a term t of the same type as $t_1, ..., t_n$ which does not unify with any of these terms. Now we specify the i^{th} pattern (for $1 \leq i \leq n$) as $f(t_i, s_1, .., s_m)$, where s_j is a or b depending upon whether x_j or $\neg x_j$ occurs in φ_i. If neither occur in φ_i then s_j is a variable. Observe that the size of this pattern set is polynomial in the size of $\varphi_1, ..., \varphi_n$. With this construction, we will now show that determining whether $f(x_0, ..., x_m)$ possesses an index is equivalent to determining whether $\varphi_1 \wedge \cdots \wedge \varphi_n$ is *not* satisfiable.

First we transform the above pattern set into a set of constrained patterns. Following the transformation the $(n+1)^{\text{th}}$ pattern becomes

$$\{f(x_0, ..., x_m) | (x_0 \neq t_1 \vee \varphi_1) \wedge \cdots \wedge (x_0 \neq t_n \vee \varphi_n)\}$$

Here we have slightly abused the notation in replacing $x_j \neq a$ by x_j and $x_j \neq b$ (i.e. $x_j = a$ by type discipline) by $\neg x_j$. We now show that x_0 is *not* an index of the above constrained term iff $\varphi_1 \wedge \cdots \wedge \varphi_n$ is satisfiable. Suppose that $\varphi_1 \wedge \cdots \wedge \varphi_n$ is satisfied by a truth assignment $\mathcal{T}$. Then each φ_i is satisfied by $\mathcal{T}$. Consider the instance of the term $f(x_0, t_1, ..., t_m)$, where t_i is b or a respectively, depending upon whether $\mathcal{T}(x_i)$ is $false$ or $true$. Clearly, this term is an instance of the constrained term, yet x_0 is a variable in it – which means that x_0 is *not* an index. For proving the converse, suppose that x_0 is not an index, i.e., there is an instance of the constrained term that does not instantiate x_0. Then, the substitutions for each $x_1, ..., x_m$ must satisfy φ_1 through φ_n. This implies that $\varphi_1 \wedge \cdots \wedge \varphi_n$ must be satisfiable, thus completing the proof. ■

Since the index computation algorithm is very hard in general, we need to examine heuristics that can speed up the process in most cases. Two such heuristics are described in [25].

5 Space and Matching Time Complexity

We now examine upper and lower bounds on the space and matching time complexity of adaptive tree automata for several classes of patterns. Since the traversal order itself is a parameter here, we first need to clarify what we mean by upper and lower bounds. By an upper bound, we refer to an upper bound obtained by using the best possible traversal for a set of patterns, i.e., a traversal that minimizes space (or time, as the case may be). The rationale for this definition is that for every set of patterns, there exist traversal orders that can result in the worst possible time or space complexity. Clearly, it is not interesting to talk about the upper bound on size of the automaton obtained using such a (deliberately chosen) non-optimal traversal order. Our lower bounds refer to the lower bounds obtained for any possible traversal order.

Figure 6 summarizes our results. The proof on upper bound on space follows from the result of [24]. Since a left-to-right traversal is simply a special case of an adaptive traversal, the fact that there always exists a left-to-right traversal order with an automaton size less than $O(\prod_{i=1}^{n} |l_i|)$ implies the existance of an adative traversal with these bounds. We now present the details of lower bound proofs. The space bounds given in this section are all independent of the traversal order and are established using flat patterns that all have a root symbol f with arbitrarily large arity. For the purposes of building either the smallest size automaton or one that does matching in the shortest possible time, flat patterns are equivalent to a set of patterns having a common prefix u whose fringe size equals arity of f. This is because every position within the common prefix u will be an index and hence by theorem 10, an automaton of smallest size (and matching time) can be obtained by first visiting all these positions. The structure of the automaton after

Class of Patterns	Lower bound on space	Upper bound on space	Lower bound on time	Upper bound on time
Unambiguous, no priority	$\Omega(2^{\sqrt{\alpha}})$	$O(\prod_{i=1}^{n}\lvert l_i\rvert)$	$\Omega(\alpha)$	S
Unambiguous, with priority	$\Omega(\alpha^{n-1})$	$O(\prod_{i=1}^{n}\lvert l_i\rvert)$	$\Omega(S)$	S
Ambiguous	$\Omega(\alpha^{n-1})$	$O(\prod_{i=1}^{n}\lvert l_i\rvert)$	$\Omega(S)$	S

Notation
l_i : i^{th} pattern
n : Number of Patterns
S : Total number of nonvariable symbols in patterns
α : Average number of nonvariable symbols in patterns

Figure 6: Space and matching time complexity of adaptive automata.

$$\begin{bmatrix} a & a & a & \_ & \_ & \_ & a \\ b & \_ & \_ & a & a & \_ & \_ \\ \_ & b & \_ & b & \_ & a & \_ \\ \_ & \_ & b & \_ & b & b & \_ \end{bmatrix}$$

Figure 7: Example Matrix for $n = 4$

$$\begin{bmatrix} a & a & \_ & \_ & \_ & a \\ b & \_ & b & \_ & a & \_ \\ \_ & b & \_ & b & b & \_ \end{bmatrix} \quad \begin{bmatrix} \_ & \_ & a & a & \_ & \_ \\ b & \_ & b & \_ & a & \_ \\ \_ & b & \_ & b & b & \_ \end{bmatrix}$$

Figure 8: Matrices representing $\mathcal{L}_u$ for states reached by transitions on a and b.

this prefix is examined will be identical to that obtained for flat patterns after visiting the root symbol.

5.1 Unambiguous, Unprioritized Patterns

Consider a set of n flat patterns from the alphabet $\{f, a, b\}$ and variables. Since all flat patterns have the same root symbol f, we need only specify the arguments. Therefore the n patterns are represented by a matrix of n rows, where the i^{th} row lists the arguments of f in the i^{th} pattern. Each column has at most one occurrence of a, at most one occurrence of b and the rest are all '_'s. For each pair of patterns l and l', there is at least one column wherein l and l' have different nonvariables and so the system is unambiguous. Figure 7 shows such a matrix that represents the four patterns $f(a, a, a, \_, \_, \_, a)$, $f(b, \_, \_, a, a, \_, \_)$, $f(\_, b, \_, b, \_, a, \_)$, $f(\_, \_, b, \_, b, b, \_)$. Note that each row in the matrix contains $O(n^2)$ elements, and so the size of the matrix is $O(n^3)$.

In order to simplify the proof in this case, we will consider only those parts of the automaton reachable without following any $\neq$ transition. Let $P(n)$ denote an instance of a problem with n such patterns. Denote by $S(n)$ the size of the smallest automaton for matching $P(n)$. Suppose that the automaton chooses some position p (which will simply be a column in the matrix) to inspect. Now there are two cases to consider, depending upon whether one or two patterns have a nonvariable at p.

Case 1: Column p contains only one nonvariable: It is clear in this case that on a positive transition (i.e., transition on inspecting an a or b), we will be once again left with the same $\mathcal{L}_u$ without any reduction in the problem size. This is because the resultant problem is represented by the matrix obtained by deleting column p from the original matrix. The matrix so obtained still represents a problem of size n. For instance, in Figure 7 if we choose column 7 for inspection then we are left with the problem of building an automaton to match on the basis of the first six positions of each pattern. In other words, we are left with a matrix obtained from that of Figure 7 by deleting the last column and hence the problem is still an instance P(4) and hence $S(4)$ is the size of the smallest automaton.

Case 2: Column p contains two nonvariables: In this case, based on the symbol seen in the column we can now partition the n patterns into two sets each consisting of $n-1$ patterns. Both these sets are represented by matrices that are obtained by deleting the column p and one of the rows that contained a nonvariable at p. It is easy to see that each of these matrices represent $P(n-1)$ and hence the smallest matching automaton has the size $S(n-1)$. In the above example, inspecting position 2 results in the pattern sets $\{1,2,4\}$ and $\{2,3,4\}$ as shown in Figure 8. Hence:

$$S(n) = 2 * S(n-1)$$

whose solution is $\Omega(2^n)$.

Recall that the arity of f is $O(n^2)$, and that we use flat patterns in the proof to denote patterns with a common prefix u with fringe size equal to the arity of f. In order to have a fringe size of $O(n^2)$, the prefix must have size $O(n^2)$. Thus the average size of the patterns (α) equals n^2 and so we have:

Theorem 21 *Lower bound on space required by adaptive automata for unambiguous unprioritized patterns is* $\Omega(2^{\sqrt{\alpha}})$.

5.2 Unambiguous Prioritized Patterns

To derive lower bound on the size of the automaton in this case, consider the following set of flat patterns with textual order priority:

$$f(c^m(a), x_2, x_3, ..., x_n), f(x_1, c^m(a), x_3, ..., x_n), ..., f(x_1, ..., x_{n-1}, c^m(a))$$

In these patterns, $c^m(a)$ is an abbreviation for the term $c(c(\cdots(c(a)\cdots)$ that contains m occurrences of c. Denote by $S(n)$ the size of the automaton for patterns of the above form. We claim that the smallest size automaton is obtained by first inspecting all the c's and then the a in the first column, then those in the second column and so on. This because each position examined by such an automaton is an index by Theorem 19. Hence it follows by Theorem 10 that this automaton is no larger than any other automaton. Now consider the first m states of the automaton that correspond to inspecting all the c's in the first column. Each of these states has a transition on $\neq$ that is taken on seeing a symbol different from c. Each of these transitions lead to a state that is the root of an automaton for the remaining $n-1$ patterns. Therefore:

$$S(n) \geq mS(n-1)$$

with $S(1) = 1$. (Recall that we use breadth as the measure of size.) This means $S(n) = \Omega(m^{n-1})$. As for sufficiently large m, $m = O(\alpha)$. So, we conclude:

Theorem 22 *Lower bound on space required by adaptive automata for unambiguous prioritized patterns is* $\Omega(\alpha^{n-1})$.

5.3 Ambiguous Patterns

Consider again the set of patterns used in section 5.2. Now assume that there is no priority among these patterns. As all patterns match $f(c^m(a), c^m(a), \ldots, c^m(a))$ the system is ambiguous. Observe that the automaton A for this set of patterns must report all matches since there is no priority relationship among the patterns. We now show how we can obtain an automaton A' from A for matching the prioritized system described in section 5.2. To obtain A' we simply change the *match* annotation on the final states of A so that A' announces a match for the pattern with the highest priority among those for which a match is declared by A. It is easy to see that A' is an automaton for prioritized patterns and also that it is no larger than A. Therefore by Theorem 22,

Theorem 23 *Lower bound on space required by adaptive automata for ambiguous patterns is* $\Omega(\alpha^{n-1})$.

5.4 Matching Time

We analyze the matching time of adaptive automata in this section. Herein we derive both upper and lower bound on the matching time. We begin our discussion with the upper bound on the matching time. We would like to recall our earlier remarks regarding using $\preceq$, a partial order, as our main approach to comparing matching time of different automata. However, to give a quantitative measure of work involved in matching, we use average path lengths of "best possible" automata in the ordering given by $\preceq$.

It is clear that an upper bound on the matching time is given by the length of the longest root-to-leaf path in the automaton. Now observe that each state in a given path inspects distinct positions and this position must be a nonvariable position in at least one pattern. Hence the length of the longest path can never be more than $O(S)$. (Recall that S is the sum of sizes of all patterns.) Therefore an upper bound on the matching time is $O(S)$.

For a lower bound on matching time for unambiguous, unprioritized patterns, consider a set of *strongly sequential patterns*. For such patterns, every prefix of any pattern possesses an index. By Theorem 10, the automaton with smallest time is one that examines indices in every state. It is easy to see that in such a case, there is exactly one final state in the automaton corresponding to a match for each pattern. Moreover, all and only the nonvariable symbols in a pattern are visited on the path to a final state announcing a match for it. Therfore, the average path length of this automaton (which is our measure of matching time) is no less than the average over the sizes of the patterns, i.e., $\Omega(\alpha)$.

In the case of prioritized or ambiguous patterns the lower bound can be tightened. This is based on the following observations. In case of prioritized patterns, we must eliminate the possibility for a match for higher priority patterns before looking for a match for a lower priority pattern. This means paths leading to matches for a lower priority pattern can be substantially longer than the pattern size. Similarly, for ambiguous patterns, the need to announce all matching patterns can make lengths of matching paths for a pattern larger than the size of the pattern being matched.

We first consider the case of unambiguous prioritized patterns. We use the example presented in section 5.2 to derive the lower bound in this case. Observe that every position inspected by the automaton constructed in section 5.2 is an index. This means, by Theorem 10, the matching time for this automaton is the smallest. Hence the lower bound on matching time is given by the average path length of this automaton. We compute this quantity as follows. Let $T(n)$ denote the sum of lengths of all root-to-leaf paths in the automaton. Observe that each of the first m states that inspect the c's in the first pattern has a $\neq$ branch. These branches lead to a subautomaton that matches the remaining $n-1$ patterns and hence the sum of path lengths in each of these automata is given by $T(n-1)$. For the subautomaton S_i on the ith $\neq$ branch, the sum of path lengths from root to leaves in S_i is given by

$$T(n-1) + i * (\text{number of final states in } S_i)$$

The second term in the above expression accounts for the fact that the length i from the root of the automaton to the root of S_i is added to the length of every path to a final state in S_i. Noting that the number of final states (which is same as the breadth or space complexity) in S_i is $O(m^{n-2})$, and summing the path lengths over all the subautomata and the state announcing a match for the first pattern, we have

$$T(n) = m + \Sigma_{i=1}^{m}(T(n-1) + i * O(m^{n-2})) = m + mT(n-1) + O(m^n)$$

with $T(1) = m$. It can be easily checked by substitution that $T(n)$ is $\Omega(nm^n)$. Thus the average path length, given by $T(n)/O(m^{n-1})$ is $\Omega(mn) = \Omega(S)$.

The lower bound on matching time for ambiguous patterns can be obtained from that of the unambiguous prioritized patterns using arguments similar to that found in section 5.3. It is quite straightforward to apply the arguments found in the proof of size-complexity to matching time complexity. In particular, we can construct an automaton for prioritized patterns from that obtained for ambiguous, unprioritized patterns. This construction assures that the matching time complexity for ambiguous patterns can never be smaller than that for unambiguous prioritized patterns.

6 Minimizing Space using DAGs

In section 3, we developed several powerful techniques to reduce the space and matching time requirement of the adaptive automata. However the lower bound results established in the previous section indicates that the size of the automaton is likely to be very large. It appears (from the proofs of lower bounds) that the main reason for the exponential space requirement is the use of tree structure in representing the automaton. Lack of sharing in trees results in duplication of functionally identical subautomata leading to wastage of space. A natural solution to this problem is to implement sharing with the help of dag structure (instead of tree). We develop this solution in this section.

An obvious way to achieve sharing is to use standard FSA minimization techniques. A method based on this approach first constructs the automaton (using algorithm *Build*) and then converts it into a (optimal) dag. However, the size of the tree automaton can be exponentially larger than that of the dag automaton. For instance, the tree automaton constructed in section 5.2 has exponential size, but the corresponding dag automaton is linear! Therefore use of FSA minimization technique is bound to be very inefficient. To overcome this problem we must construct the dag automaton without generating its tree structure first. This means we must identify equivalence of two states *without even generating* the subautomata rooted at these states. Suppose we are able to identify all such pairs of equivalent states then the optimal dag automaton can be built directly for any set of patterns. We now propose a solution to this problem for the general case of adaptive automata. This important problem of directly building an optimal automata has remained open, even in the restricted context of left-to-right traversals [9].

Central to our construction (of dag automaton) is a technique that detects equivalent states based on the representative sets. Consider two prefixes u_1 and u_2 that have the same representative set $\overline{\mathcal{L}_u}$. Suppose that u_1 and u_2 differ only in those positions where every pattern in $\overline{\mathcal{L}_u}$ has a variable. Since such positions are irrelevant for determining a match, these two prefixes are equivalent. On the other hand, it can also be shown that if they have different representative sets or differ in any other position then they are not equivalent. Based on this observation, we define the relevant prefix of u as follows. Let $p_1, p_2, \ldots, p_k$ denote (all of the) positions in u such that for each p_i there is at least one pattern in $\overline{\mathcal{L}_u}$ that has a variable at p_i and all other patterns in $\overline{\mathcal{L}_u}$ have a variable either at p_i or above it. The relevant prefix of u is then

$$u[p_1 \leftarrow\neq][p_2 \leftarrow\neq]\cdots[p_k \leftarrow\neq]$$

For instance, the prefixes corresponding to different states marked 's' in Figure 3 are different, but they all have the same relevant prefix $f(x, \neq, b)$. By showing that two states are equivalent iff the corresponding relevant prefixes are identical we establish:

Theorem 24 *The automaton obtained by merging states with identical relevant prefixes is optimal.*

Proof: First we need to show that the states merged as above are indeed equivalent. Note that the condition 2 (see page 16) on the selection function at a state v requires that *Build* select the next position only based on portions of the prefix that are relevant for identifying a match at one of the descendents of v. This implies that the selection function will choose the same position to inspect for any two states with the same relevant prefix. It is also easy to see that if two states have the same relevant prefix, then the corresponding children of the two states will also have identical relevant prefixes. This implies that the structure of the automaton below any pair of states with the same relevant prefix will be identical. Therefore two such states are equivalent.

Now we need to show that no two states v_1 and v_2 with distinct relevant prefixes u_1 and u_2 are equivalent. There are two cases to consider, depending upon whether $\overline{\mathcal{L}_{u_1}}$ and $\overline{\mathcal{L}_{u_2}}$ are identical or not. If they are not identical, let $l \in \overline{\mathcal{L}_{u_1}}$ and $l \notin \overline{\mathcal{L}_{u_2}}$. Then, by the properties of representative set (and correctness of the matching automaton), there is a path from v_1 to a matching state for l. On the other hand, there is no such path from v_2 and hence v_1 and v_2 are not equivalent.

In the second case ($\overline{\mathcal{L}_{u_1}} = \overline{\mathcal{L}_{u_2}}$). Since $u_1 \neq u_2$, $\exists p$ $root(u_1/p) \neq root(u_2/p)$. Since the representative sets are the same, u_1 and u_2 cannot have different nonvariable symbols at p. Hence one of these relevant prefixes, say u_1, has a nonvariable symbol at p and the other has a variable at p. If this nonvariable is $\neq$, every pattern in the representative set must contain a variable at p. The definition of relevant prefix then implies that $root(u_2/p)$ must also be $\neq$. Since we assumed that u_1 and u_2 differ at p, this is also not possible. Therefore $root(u_1/p)$ must be a nonvariable symbol other than $\neq$. Let l be an pattern in the representative set such that

$root(l/p)$ is a nonvariable. Now, there must be a path from v_1 to a matching state for l, and the symbol at p is not examined on this path (as it has already been examined in the path reaching v). In contrast, the symbol at p is examined on every path from v_2 to a matching state for l. Therefore v_1 and v_2 are not equivalent. ∎

Merging equivalent states as described above can substantially reduce the space required by the automata, e.g., the tree automaton in Figure 3 has 25 states which can now be reduced to 16 by sharing. Also recall that for the patterns in Figure 7, parts of the automaton reached by positive transitions alone (i.e., without considering parts of automata that are reached through $\neq$ transitions) is exponential. We can show that by sharing states this part of the automaton will become polynomial! Similarly, the size of the automaton constructed in section 5.2 for unambiguous, prioritized patterns will become linear in the size of patterns rather than being exponential. (This is because all the states that are reached on $\neq$ transitions from the first m states will be equivalent, and thus there will be only one subautomaton of size $S(n-1)$ instead of m.)

6.1 Impact of DAGs on Space and Matching Time Complexity

Observe that sharing affects space requirements alone. Therefore all our earlier results not directly related to space continue to hold for dags as well. In what follows we discuss the impact of dags on some of the results established earlier regarding space.

We can show that the upper bound on size of dag automata is $O(2^n S)$ which is much smaller than the corresponding bound $O(\prod_{i=1}^{n} |l_i|)$ for tree automata. To prove this result, consider a dag automaton based on left-to-right traversal of patterns. Consider the relevant prefixes of any two states in this automaton with the same representative set $\overline{\mathcal{L}_u}$. Since the symbols are visited in pre-order, one of these prefixes u_1 must be an instance of the other prefix u_2. Extending this argument to all states $s_1, ..., s_k$ with the same representative set, we see that there is a total order among these prefixes (given by the above-mentioned instance-of relation). This means that the number of such prefixes (and hence the number of such states) is bounded by the size of the largest prefix, which is in turn bounded by S. This bound, in conjunction with the fact that there are at most 2^n different representative sets, yields the bound of $O(S * 2^n)$.

We can also establish a lower bound of $O(2^\alpha)$ for ambiguous patterns. For this proof, consider n flat patterns of the form

$$\begin{aligned} l_1 &= f(a, x_2, x_3, ..., x_n) \\ l_2 &= f(x_1, a, x_3, ..., x_n) \\ \vdots & \quad \vdots \quad \vdots \\ l_n &= f(x_1, ..., x_n - 1, a) \end{aligned}$$

By our earlier remarks on flat patterns, note that the average size α of these patterns is $O(n)$. There is no priority among the patterns, so the automaton is required to report all patterns that match a given term. It is clear that any term $f(t_1, ..., t_n)$ matches the set of patterns $\{p_{i1}, ..., p_{ik}\}$ whenever $t_{i1} = t_{i2} = ...t_{ik} = a$. There are 2^n such sets, each of which must correspond to a state in the automaton and hence the bound.

For unambiguous patterns, it is not clear whether the lower bound on size is exponential. For instance, it appears that the patterns used in the lower bound proof on size of tree automata for unambiguous patterns, possess a polynomial-size dag automaton. In the example used to establish lower bound on space for tree automaton for unambiguous prioritized patterns, observe that the first m states reached by $\neq$ transitions (i.e., states reached on inspecting any symbol other than c in the first column) are all equivalent. By sharing all these states, we get the recurrence relation $S(n) = m + S(n-1)$, whose solution is $O(S)$. Reasoning about lower bounds becomes extremely complicated for dags since it is difficult to capture behavior of sharing formally.

All the greedy strategies as well as the strategy of selecting indices can, in some cases, increase the space of dag automata. This increase occurs typically in contrived examples. In practice, we should use some of the greedy strategies and the index selection strategy to reduce space and matching time. Sharing of states then provides additional opportunities for further reduction in space required.

7 Concluding Remarks

In this paper we studied pattern matching with adaptive automata. We first presented a generic algorithm for their construction and then discussed how to improve space and matching time by synthesizing traversal orders. We showed that a good traversal order selects indices whenever possible and uses one of the greedy strategies otherwise. Although the greedy strategies may sometimes fail, it appears from the complexity of the counter examples that such failures may be rare. For functional programming, we synthesized a traversal $S(T)$ from a monotonic traversal T. Since using $S(T)$ does not affect termination properties, the programmer can assume T whereas an implementation can benefit from significant improvements in space and matching time.

Our lower bound results indicate that the size of an adaptive automaton, when represented as a tree, can be quite large. Therefore we developed an orthogonal approach to space minimization by sharing equivalent states. Note that even the index selection strategy may fail to improve space of dag automata. This occurs because index selection may adversely affect the way in which descendants of a state can be shared. Since it is difficult to predict sharing among descendant states, the possibility of improving space without using indices does not appear to be practical. So the best approach is to use all the strategies in section 3 and use sharing as an additional source of space optimization over tree automata.

Our work clearly brings forth the impact of typing in prioritized pattern matching. We have shown that several important problems in the context of pattern matching are unlikely to have polynomial-time algorithms for typed systems whereas we have given polynomial-time algorithms for them in untyped systems. This raises the question whether it is worthwhile to consider typing for pattern matching. It is not clear how often typing information can be used to find an index (or to determine that a pattern does not belong to $\overline{\mathcal{L}_u}$) which cannot be found otherwise. On the other hand there is a significant penalty in terms of computational effort for both these problems if we use typing information.

We have left open two aspects of the problem that may be important in certain applications. First, our algorithms are mainly suited for an environment wherein the patterns do not change very frequently. This is because the traversal order itself can change when patterns are changed.

Second, the algorithms cannot handle variables in the term to be matched. If we can handle variables, then the automaton can be used for fast unification, which has many applications such as Prolog compilers and theorem provers. We are currently investigating techniques to extend our algorithms to address these concerns.

References

[1] L. Augustsson, A compiler for Lazy ML, *LISP and Functional Programming Conference*, 1984.

[2] L. Augustsson, Compiling Pattern Matching, *Functional Programming and Computer Architecture*, 1985.

[3] R.Burstall, D.MacQueen and D.Sanella, HOPE: An Experimental Applicative Language, *International LISP Conference*, 1980.

[4] J. Christian, Flatterms, Discrimination Nets and Fast Term Rewriting, *Journal of Automated Reasoning, 10, 1993.*

[5] D. Comer and R. Sethi, Complexity of Trie Index Construction, *Foundations of Computer Science*, 1976.

[6] T.A. Cooper and N. Wogrin, Rule-based Programming with OPS5, *Morgan KaufMann, 1988.*

[7] N. Dershowitz and J.P. Jouannaud, Rewrite Systems, *Handbook of Theoretical Computer Science*, Ch. 6, Vol II, North-Holland, 1990.

[8] Garey and Johnson, Computers and Intractability, A guide to the theory of NP-Completeness., W.H. Freeman, San Francisco, 1979.

[9] A. Graf, Left-to-Right Tree Pattern Matching, *Rewriting Techniques and Application*, 1991.

[10] R.Harper, R.Milner and M.Tofte, The Definition of Standard ML, *Report ECS-LFCS-88-62*, Laboratory for Foundations of Computer Science, University of Edinburgh, 1988.

[11] P. Henderson, Functional Programming: Application and Implementation, *Prentice-Hall*, 1980.

[12] C.M. Hoffmann and M.J. O'Donnell, Pattern Matching in Trees, *Journal of the ACM, 29, 1, 1982 .*

[13] P. Hudak et al, Report on the Programming Language Haskell, Version 1.1, Yale and Glasgow Universities, 1991.

[14] G. Huet and J.J. Levy, Computations in Nonambiguous Linear Term Rewriting Systems, *Tech. Rep. No. 359*, IRIA, Le Chesney, France, 1979.

[15] G. Kahn and G. Plotkin, Domaines Concretes, *Tech Report 336, IRIA Laboria, Le Chesnay, France, 1978.*

[16] J.R. Kennaway, The Specificity Rule for Lazy Pattern Matching in Ambiguous Term Rewriting Systems, *European Symposium on Programming*, 1990.

[17] A. Laville, Lazy Pattern Matching in the ML Language, *Foundations of Software Technology & Theoretical Computer Science*, 1987.

[18] A. Laville, Implementation of Lazy Pattern Matching Algorithms, *European Symposium on Programming*, 1988.

[19] L. Maranget, Compiling Lazy Pattern Matching, *Lisp and Functional Programming, 1992.*

[20] J. McDermott and C.L. Forgy, Production System Conflict Resolution Strategies, in D. Waterman, D. Hayes-Roth and D. Lenat (eds.), *Pattern-Directed Inference Systems, Academic Press, 1978.*

[21] M.J. O'Donnell, Equational Logic as a Programming Language, MIT Press, 1985.

[22] K. Owen, S. Pawagi, C. Ramakrishnan, I.V. Ramakrishnan, R.C. Sekar, Fast Parallel Implementation of Functional Languages - The EQUALS Experience, *Lisp and Functional Programming*, 1992.

[23] L. Puel and A. Suarez, Compiling Pattern Matching by Term Decomposition, *Lisp and Functional Programming '90.*

[24] Ph. Schnoebelen, Refined Compilation of Pattern Matching for Functional Languages, *Science of Computer Programming*, 1988.

[25] R.C. Sekar, R. Ramesh and I.V. Ramakrishnan, Adaptive Pattern Matching, *International Conference on Automata, Languages and Programming, 1992.*

[26] P. Wadler, Efficient Compilation of Pattern Matching, in S.L. Peyton-Jones, Ed, The Implementation of Functional Programming Languages, Prentice Hall, 1987.