International Journal of Foundations of Computer Science
Vol. 16, No. 3 (2005) 499–510
© World Scientific Publishing Company

THE GENERALIZATION OF GENERALIZED AUTOMATA: EXPRESSION AUTOMATA*

YO-SUB HAN†
Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong SAR
emmous@cs.ust.hk

DERICK WOOD†
Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong SAR
dwood@cs.ust.hk

Received 29 November 2004
Accepted 21 February 2005
Communicated by Kai Salomaa

We explore expression automata with respect to determinism and minimization. We define determinism of expression automata using prefix-freeness. This approach is, to some extent, similar to that of Giammarresi and Montalbano's definition of deterministic generalized automata. We prove that deterministic expression automata languages are a proper subfamily of the regular languages. We close by defining the minimization of deterministic expression automata.

Keywords: Expression automata, state elimination and prefix-freeness.

1. Introduction

Recently, there has been a resurgence of interest in finite-state automata that allow more complex transition labels. In particular, Giammarresi and Montalbano [3] have studied **generalized automata** (introduced by Eilenberg [2]) with respect to determinism. Generalized automata have strings (or **blocks**) as transition labels rather than merely characters or the null string. (They have also been called string or lazy automata.) Generalized automata allow us to more easily construct an automaton in many cases. For example, given the reserved words for C++ programs, construct a finite-state automaton that discovers all reserved words that appear in a

*A preliminary version of this paper appeared in the *Proceedings of the 9th International Conference on Implementation and Application of Automata*, CIAA'04, Ontario, July 22-24, 2004.
†The authors were supported under the Research Grants Council Competitive Earmarked Research Grant HKUST6197/01E.

specific C++ program or program segment. The use of generalized automata makes this task much simpler.

It is well known that generalized automata have the same expressive power as traditional finite-state automata. Indeed, we can transform any generalized automaton into a traditional finite-state automaton using **state expansion.** Giammarresi and Montalbano, however, took a different approach by defining **deterministic generalized automata (DGAs)** directly in terms of a local property which we introduce in Section 4.

Our goal is to re-examine the notion of **expression automata;** that is, finite-state automata whose transition labels are regular expressions over the input alphabet. We define **deterministic expression automata (DEAs)** by extending the applicability of prefix-freeness.

We first define traditional finite-state automata and generalized automata and their deterministic counterparts in Section 2 and formally define expression automata in Section 3. In Section 4, we define determinism based on prefix-freeness and investigate the relationship between deterministic expression automata and prefix-free regular languages. Then, we consider minimization of deterministic expression automata in Section 5.

2. Preliminaries

Let Σ denote a finite alphabet of characters and Σ^* denote the set of all strings over Σ. A language over Σ is any subset of Σ^*. The character $\emptyset$ denotes the empty language and the character λ denotes the null string. Given two strings x and y in Σ^*, x is said to be a **prefix** of y if there is a string w such that $xw = y$. Given a set X of strings over Σ, X is **prefix-free** if no string in X is a prefix of any other string in X.

A traditional finite-state automaton A is specified by a tuple $(Q, \Sigma, \delta, s, F)$, where Q is a finite set of states, Σ is an input alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is a (finite) set of transitions, $s \in Q$ is the start state and $F \subseteq Q$ is a set of final states. Given a transition (p, a, q) in δ, where $p, q \in Q$ and $a \in \Sigma$, we say p has an out-transition and q has an in-transition. Furthermore, p is a source state of q and q is a target state of p. A string x over Σ is accepted by A if there is a labeled path from s to a final state in F that spells out x. Thus, the language $L(A)$ of a finite-state automaton A is the set of all strings spelled out by paths from s to a final state in F. Automata that have only **useful** states, that is, each state appears on some path from the start state to some final states are called **trim** or **reduced** [2, 7].

Eilenberg [2] introduced **generalized automata,** an extension of traditional finite-state automata by allowing strings on the transitions. A generalized automaton A is specified by a tuple $(Q, \Sigma, \delta, s, F)$, where Q is a finite set of states, Σ is an input alphabet, $\delta \subseteq Q \times \Sigma^* \times Q$ is a finite set of block transitions, $s \in Q$ is the start state and $F \subseteq Q$ is a set of final states. Giammarresi and Montalbano [3] defined a deterministic generalized automaton using a local notion of **prefix-freeness.** A generalized automaton A is deterministic if, for each state q in A, the following two conditions hold:

1. The set of all blocks in out-transitions from q is prefix-free.
2. For any two out-transitions (q, x, p) and (q, y, r) from q, if $x = y$, then we require that $p = r$.

Note that Giammarresi and Montalbano do not require condition 2 and, as a result, some DGAs are nondeterministic.

Since regular languages are sets of strings, we can apply the notion of prefix-freeness to such sets.

Definition 1 *A (regular) language L over an alphabet Σ is prefix-free if, for all distinct strings x and y in L, x is not a prefix of y and y is not a prefix of x. A regular expression α is prefix-free if $L(\alpha)$ is prefix-free.*

Lemma 1 *A regular language L is prefix-free if and only if a trim deterministic finite-state automaton (DFA) A for L has no out-transitions from any final states.*

Proof. We first prove that if A is a trim DFA that has no out-transitions from any final states, then $L(A)$ is prefix-free. Now, each string in $L(A)$ is spelled out by a path in A from the start state s to a final state $f \in F$. If $|L(A)| = 1$, there is only one path in A; therefore, $L(A)$ is prefix-free.

On the other hand, if $|L(A)| > 1$, then assume that there are two distinct strings x and y in $L(A)$ such that x is a prefix of y; that is, $y = xz$, where $z \neq \lambda$. Now, x is spelled out by a path from s to some $f \in F$ and because A is a DFA, the prefix of y of length $|x|$ is also spelled out by the same path. However, since $|z| \geq 1$, there is an out-transition (f, z_1, q_1) from state f to some state q_1, where z_1 is the first character of z. Since A is trim, there must be a transition sequence $(f, z_1, q_1) \cdots (q_{m-1}, z_m, q_m)$, for some $m \geq 1$, such that $z_1 \cdots z_m = z$ and $q_m \in F$. In other words, there is an out-transition from f — a contradiction. Thus, $L(A)$ is prefix-free.

Conversely, if L is prefix-free, then a trim DFA A for L has no out-transitions from any final states. Assume that A has an out-transition from a final state f. Then, a path from s to f defines an accepted string x and the out-transition from f also leads to a final state $g \in F$ that defines another accepted string xy, where $|y| \geq 1$. Since the out-transition from f is labeled with a character over Σ, $|xy| > |x|$. Thus, A accepts both x and xy, where $y \neq \lambda$; hence, L is not prefix-free — a contradiction. Therefore, final states have no out-transitions. □

3. Expression automata

It is well known that regular expressions and (deterministic) finite-state automata have exactly the same expressive power [5, 9]. A finite-state automaton allows only a single character in a transition and a generalized automaton [2] allows a single string, possibly the null string, in a transition. It is natural to extend this notion to allow a regular expression in a transition, since a character and a string are also regular expressions. This concept was first considered by Brzozowski and McCluskey, Jr. [1] to compute regular expressions from finite-state automata.

Definition 2 *An* **expression automaton** *A is specified by a tuple $(Q, \Sigma, \delta, s, f)$, where Q is a finite set of states, Σ is an input alphabet, $\delta \subseteq Q \times \mathcal{R}_\Sigma \times Q$ is a finite* *set of expression transitions, where $\mathcal{R}_\Sigma$ is the set of all regular expressions over Σ, $s \in Q$ is the start state and $f \in Q$ is the final state. (Note that we only have one final state.) We require that, for every pair p and q of states, there is exactly one expression transition (p, α, q) in δ, where α is a regular expression over Σ.*

We can also use the functional notation $\delta: Q \times Q \rightarrow \mathcal{R}_\Sigma$ that gives the equivalent representation. An expression transition (p, α, q) gives $\delta(p, q) = \alpha$. Note that δ contains exactly $|Q|^2$ transitions, one transition for each pair of states, and whenever $(p, \emptyset, q)$ is in δ, for some p and q in Q, A cannot move from p to q directly.

We generalize the notion of accepting transition sequences to accepting expression transition sequences and accepting language transition sequences.

Definition 3 *An* **accepting expression transition sequence** *is a transition sequence of the form:*

$$(p_0 = s, \alpha_1, p_1) \cdots (p_{m-1}, \alpha_m, p_m = f),$$

for some $m \geq 1$, where s and f are the start and final states, respectively.

The second notion is an **accepting language transition sequence** *of the form:*

$$(p_0 = s, L(\alpha_1), p_1) \cdots (p_{m-1}, L(\alpha_m), p_m = f),$$

for some $m \geq 1$, where s and f are the start and final states, respectively.

We define an **expression automaton language** to be the language accepted by an expression automaton.

Lemma 2 *Every trim finite-state automaton can be converted into an equivalent trim expression automaton. Therefore, every regular language is an expression automaton language.*

Proof. Let $A = (Q, \Sigma, \delta, s, F)$ be a trim finite-state automaton. We construct a trim expression automaton $A' = (Q \cup \{f\}, \Sigma, \delta', s, f)$ from A as follows, where state f is not in Q.

1. For all p and q in Q, $(p, \emptyset, q)$ is in δ' if (p, a, q) is not in δ, for all $a \in \Sigma$. Otherwise, letting $\alpha = (a_1 + \cdots + a_r)$, where (p, a_i, q) is in δ, $1 \leq i \leq r$, and (p, a, q) is not in δ, for all $a \in \Sigma \setminus \{a_1, \ldots, a_r\}$, we observe that (p, α, q) is in δ'.
2. For all $f' \in F$, (f', λ, f) is in δ'.
3. For all $p \in Q \setminus F$, $(p, \emptyset, f)$ is in δ'.
4. For all $p \in Q \cup \{f\}$, $(f, \emptyset, p)$ is in δ'.

The construction is straightforward except for the first case. Given two states p and q, we merge all transitions of the form (p, a_i, q), for all i, $1 \leq i \leq r$, for some $r \geq 1$, to give the expression transition $(p, (a_1 + \cdots + a_r), q)$ or, equivalently, the language transition $(p, \{a_1, \ldots, a_r\}, q)$.

Clearly, $L(A) = L(A')$ since accepting transition sequences in A become accepting expression sequences in A' when a terminating λ-transition is appended to ensure acceptance. □

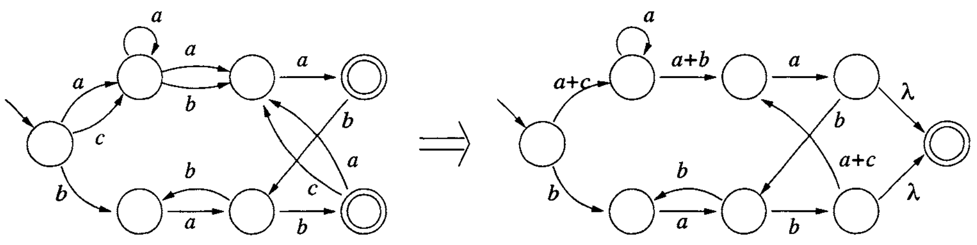


Figure 1: An example of transforming a finite-state automaton into an expression automaton. We omit all $\emptyset$-transitions between states.

We next establish that we can convert every expression automaton A into an equivalent finite-state automaton; that is combining the two results, expression automata and finite-state automata have the same expressive power. We prove this fact by constructing a regular expression α such that $L(\alpha) = L(A)$. A trim expression automaton $A = (Q, \Sigma, \delta, s, f)$ is **non-returning** if $\delta(q, s) = \emptyset$, for all $q \in Q$. Note that any trim expression automaton A can be converted into a trim non-returning expression automaton for the same language $L(A)$ by introducing a new start state s' and a transition $\delta(s', \lambda, s)$.

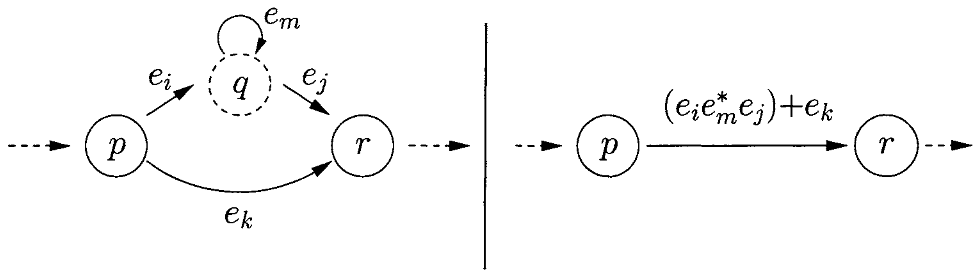


Figure 2: An example of the state elimination of a state q.

We define **state elimination** of $q \in Q \backslash \{s, f\}$ in A to be the bypassing of state q, q's in-transitions, q's out-transitions and q's self-looping transition with equivalent expression transition sequences. For each in-transition (p_i, α_i, q), $1 \le i \le m$, for some $m \ge 1$, for each out-transition (q, γ, r_j), $1 \le j \le n$, for some $n \ge 1$, and for the self-looping transition (q, β, q) in δ, construct a new transition $(p_i, \alpha_i \cdot \beta^* \cdot \gamma_j, r_j)$. Since there is always an existing transition (p, ν, r) in δ, for some expression ν, we merge the two transitions to give the bypass transition $(p, (\alpha_i \cdot \beta^* \cdot \gamma_j) + \nu, r)$. We then remove q and all transitions into and out of q in δ. We denote the resulting expression automaton by $A_q = (Q \setminus \{q\}, \Sigma, \delta_q, s, f)$ after the state elimination of q. Thus, we have established the following state elimination result:

Lemma 3 *Let $A = (Q, \Sigma, \delta, s, f)$ be a trim and non-returning expression automaton with at least three states and q be a state in $Q \setminus \{s, f\}$. Define $A_q = (Q \setminus \{q\}, \Sigma, \delta_q, s, f)$ to be a trim and non-returning expression automaton such that, for* *all pairs p and r of states in $Q \setminus \{q\}$,*

$$\delta_q(p,r) = \delta(p,r) + (\delta(p,q) \cdot \delta(q,q)^* \cdot \delta(q,r)).$$

Then, $L(A_q) = L(A)$ and A_q is trim and non-returning.

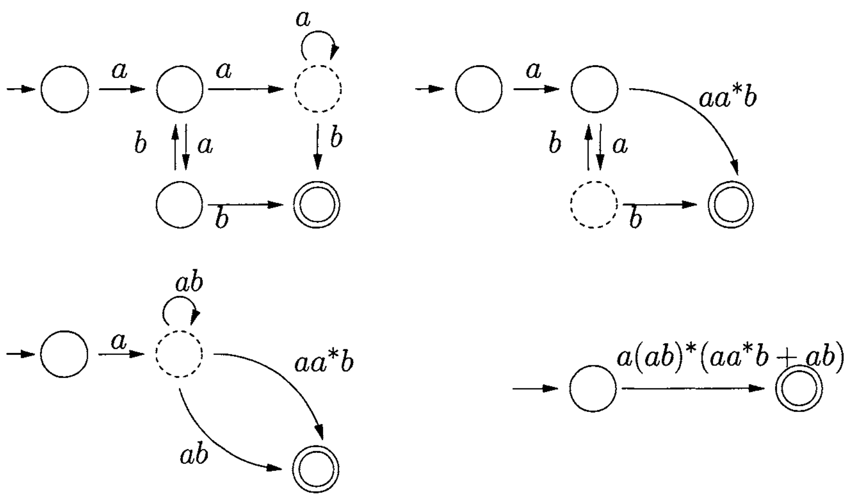


Figure 3: An expression automaton for the regular language $L(a(ab)^*(aa^*b + ab))$ and its state eliminations.

The elimination of a state q in a given trim expression automaton A preserves all the labeled paths from q's predecessors to its successors. Therefore, state elimination does not change the language accepted by A.

To complete the construction of an equivalent regular expression, we repeatedly eliminate one state at a time until $Q = \{s, f\}$. Thus, we are left with a trim and non-returning expression automaton $\bar{A}$ that has exactly two states s and f. Note that $\delta(s,s) = \emptyset$ and $\delta(f,s) = \emptyset$ since $\bar{A}$ is trim and non-returning. Thus, only the transitions $\delta(s,f)$ and $\delta(f,f)$ can be nontrivial. Hence, $L(\bar{A}) = L(\delta(s,f) \cdot \delta(f,f)^*) = L(A)$. We have established the following result:

Theorem 1 *A language L is an expression automaton language if and only if L is a regular language.*

Proof. If L is the language of an expression automaton A, then we can construct an equivalent regular expression E by state elimination such that $L = L(E)$. Furthermore, if L is a regular language, then it is an expression automaton language by Lemma 2. □

4. Deterministic expression automata

We now define **deterministic expression automata (DEAs)** and investigate their properties. A traditional finite-state automaton is **deterministic** if, for each state, the next state is uniquely determined by the current state and the current input character [9].

For an expression automaton, the situation is not as simple. When processing an input string with a given expression automaton and a given current state, we need to determine not only the next state but also an appropriate prefix of the remaining input string since each of the current state's out-transitions is labeled with a regular expression (or a regular language) instead of with a single character.

An expression automaton is deterministic if and only if, for each state p of the automaton, each two distinct out-transitions have disjoint regular languages and, in addition, each regular language is prefix-free. For example, the out-transition

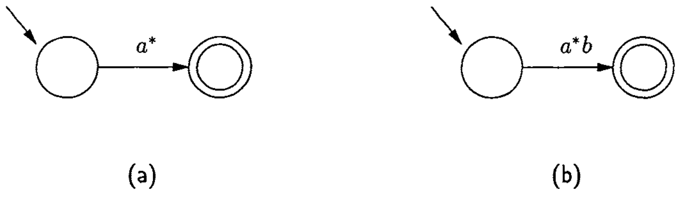


Figure 4: a. Example of non-prefix-freeness. b. Example of prefix-freeness.

of the expression automaton in Fig. 4(a) is not prefix-free; $L(a^*)$ is not prefix-free since a^i is a prefix of a^j, for all i and j such that $1 \leq i < j$. Hence, this expression automaton is not deterministic. On the other hand, the expression automaton in Fig. 4(b) is deterministic since $L(a^*b)$ is a prefix-free language. We formalize the definition as follows:

Definition 4 *An expression automaton $A = (Q, \Sigma, \delta, s, f)$, where $|Q| = m$, is* **deterministic** *if and only if the following three conditions hold:*

1. **Prefix-freeness:** *For each state $q \in Q$ and for q's out-transitions*

$$(q, \alpha_1, q_1),\ (q, \alpha_2, q_2),\ \ldots,\ (q, \alpha_m, q_m),$$

   $L(\alpha_1) \cup L(\alpha_2) \cup \cdots \cup L(\alpha_m)$ is a prefix-free regular language.

2. **Disjointness:** *For each state $q \in Q$ and for all pairs of out-transitions α_i and α_j, where $i \neq j$ and $1 \leq i,\ j \leq m$,*

$$L(\alpha_i) \cap L(\alpha_j) = \emptyset.$$

3. **Non-exiting:** *$(f, \emptyset, q)$ in δ, for all $q \in Q$.*

Lemma 4 *If a trim DEA $A = (Q, \Sigma, \delta, s, f)$ has at least three states, then, for any state $q \in Q \setminus \{s, f\}$, A_q is deterministic.*

Proof. Consider a state q in a DEA A. Let (p, α, q), (q, β, t) and (p, γ, t) be transitions in δ. Then, by the definition of DEAs, $L(\alpha)$, $L(\beta)$ and $L(\gamma)$ are prefix-free regular languages and $L(\alpha)$ and $L(\gamma)$ are disjoint from each other. We remove a state q and its transitions β. Then α and β are catenated as $\alpha\beta$, which preserves prefix-freeness. Note that $L(\alpha\beta)$ and $L(\gamma)$ are still disjoint. Therefore, A_q is deterministic; namely, state elimination for a DEA preserves determinism. □

However, the converse of Lemma 4 does not hold.

Lemma 5 *There exists a trim expression automaton A that is deterministic if and only if $L(A)$ is prefix-free.*

Proof.

$\Longrightarrow$ Assume that $A = (Q, \Sigma, \delta, s, f)$ is deterministic. Let $Q' = Q \setminus \{s, f\}$ and $m = |Q'|$. If $m = 0$, then A has exactly two states (s and f) and at most two nontrivial transitions (s, α, s) and (s, β, f) in δ. Thus, $L(\alpha^*\beta)$ is prefix-free since A is deterministic.

If $m > 0$, then apply state elimination to A for each $q \in Q'$. The resulting automaton $\bar{A}$ has exactly two states and two nontrivial transitions, (s, α, s) and (s, β, f) in δ, where $L(\alpha^*\beta)$ is a prefix-free language by Lemma 4. Since $L(A) = L(\bar{A}) = L(\alpha^*\beta)$ and $L(\alpha^*\beta)$ is a prefix-free language, $L(A)$ is a prefix-free language.

$\Longleftarrow$ Assume that $L(A)$ is prefix-free. Then, there is a prefix-free regular expression α such that $L(A) = L(\alpha)$. Using α, we can construct the expression automaton $A = (\{s, f\}, \Sigma, \{(s, \alpha, f)\}, s, f)$ such that $L(A) = L(\alpha)$. Moreover, since $L(\alpha)$ is a prefix-free language and α is the only transition in A, A is deterministic. $\square$

Lemma 5 demonstrates that the regular languages accepted by DEAs are prefix-free. Thus, DEA languages define a proper subfamily of the regular languages.

Theorem 2 *The family of prefix-free regular languages is closed under catenation and intersection but not under union, complement or star.*

These closure and nonclosure results can be proved straightforwardly.

5. Minimization of DEAs

It is natural to attempt to reduce the size of an automaton as much as possible to save space. There are well-known algorithms [4, 6] to truly minimize DFAs in that they give unique (up to a renaming of states) minimal DFAs. Recently, Giammarresi and Montalbano [3] suggested a minimization algorithm for **deterministic generalized automata** (DGAs). The technique does not, however, result in a unique minimal DGA. Given a DGA, they introduce two operations in their quest for a minimal DGA. The first operation identifies indistinguishable states similar to minimization for DFAs and the second operation applies state elimination to reduce the number of states in a DGA (at the expense of increasing the label lengths of the transitions).

We define the minimization of a DEA as the transformation of a given DEA into a DEA with a smaller number of states. Note that, for all DEAs, we can construct an equivalent simple DEA, which consists of one start state and one final state with one transition between them, from any DEA using a sequence of state eliminations. However, this sequence of state eliminations makes the regular expression of the transition more complex.

Given a trim DEA $A = (Q, \Sigma, \delta, s, f)$ and a state $q \in Q$, we define the **right language** $L_{\overrightarrow{q}}$ to be the set of strings defined by the trim DEA $A_{\overrightarrow{q}} = (Q', \Sigma', \delta', q, f)$, where $Q' \subseteq Q, \Sigma' \subseteq \Sigma$ and $\delta' \subseteq \delta$. Similarly, we define the **left language** $L_{\overleftarrow{q}}$ defined by the trim DEA $A_{\overleftarrow{q}} = (Q', \Sigma', \delta', s, q)$, where $Q' \subseteq Q, \Sigma' \subseteq \Sigma$ and $\delta' \subseteq \delta$.

We then define two distinct states p and q to be **indistinguishable** if $L_{\overrightarrow{p}} =$

$L_{\overrightarrow{q}}$. We denote this indistinguishability by $p \sim q$. Note that if $p \sim q$, then there must be a pair of indistinguishable states in the following states in a DFA. However, this property does not always hold for a DEA; Fig. 5 illustrates it.

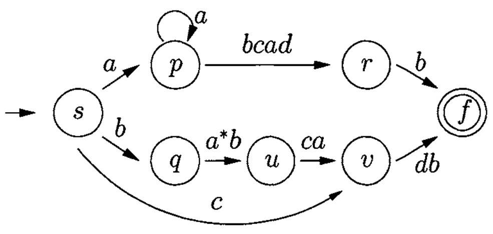


Figure 5: An example of indistinguishable states. Note that r and u are distinguishable although $p \sim q$.

Based on the notion of the right language, we define a minimal DEA as follows.
Definition 5 *A trim DEA A is minimal if all states A are distinguishable from each other.*

Thus, we minimize a DEA by merging indistinguishable states. We now explain how to merge two indistinguishable states p and q to give one state p, say. The method is simple, we first remove state q and its out-transitions and then redirect its in-transitions into state p. Once we have defined this micro-operation, we can repeat it wherever and whenever we find two indistinguishable states. Since there are only finitely many states, we can guarantee termination and minimality.

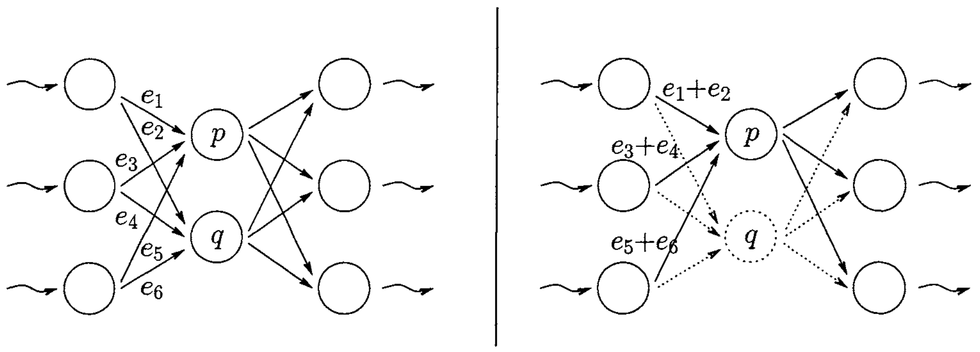


Figure 6: An example of the merging of two indistinguishable states p and q. The dotted lines show the removal of transitions.

Now we need to prove that the micro-operation on $p \sim q$ in A does not change $L(A)$. Observe that since $L_{\overrightarrow{p}} = L_{\overrightarrow{q}}$, we can remove state q and its out-transitions and redirect q's in-transitions to be in-transitions of p. Now, let $L_{\overleftarrow{p}}$ and $L_{\overleftarrow{q}}$ be the left languages of p and q. Observe that redirecting q's in-transitions to be new in-transitions of p implies that the new left language of p is now $L_{\overleftarrow{p}} \cup L_{\overleftarrow{q}}$ whereas before the redirection the left languages of p and q are $L_{\overleftarrow{p}}$ and $L_{\overleftarrow{q}}$. Moreover, since $L_{\overrightarrow{p}} = L_{\overrightarrow{q}}$, once q is removed the right language of p is unchanged. Finally, we catenate the two languages to obtain $(L_{\overleftarrow{p}} \cup L_{\overleftarrow{q}}) \cdot L_{\overrightarrow{p}} = (L_{\overleftarrow{p}} \cdot L_{\overrightarrow{p}}) \cup (L_{\overleftarrow{q}} \cdot L_{\overrightarrow{p}}) = (L_{\overleftarrow{p}} \cdot L_{\overrightarrow{p}}) \cup (L_{\overleftarrow{q}} \cdot L_{\overrightarrow{q}})$, before the removal of q. Notice that the resulting expression automaton is still deterministic.

Note that, as with DGA, we cannot guarantee that we obtain a unique minimum DEA from a given DEA. We can only guarantee that we obtain a minimal DEA. For example, the automaton in Fig. 5 can be minimized in at least two different ways as shown in Fig. 7.

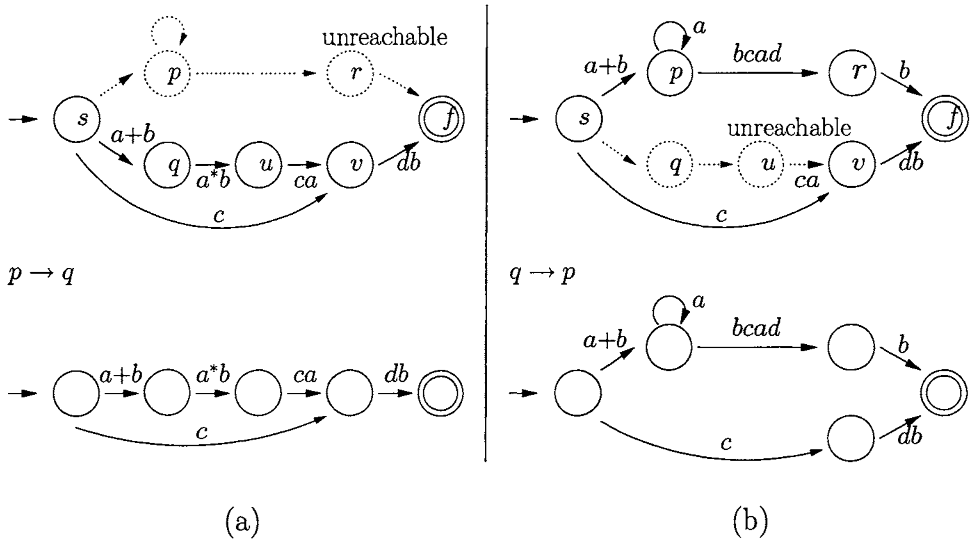


Figure 7: Two different minimal DEAs for the DEA in Fig. 5.

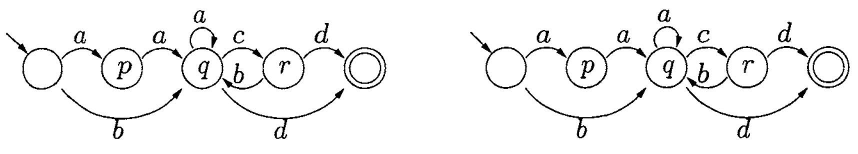
As shown in Fig. 7(a), we merge p into q and remove state r since it is unreachable after merging. In Fig. 7(b), we merge q into p and remove state u since it is unreachable. However, the second state v from q has an in-transition from s, which prevents v from being useless. The two minimizations result in two different minimal expression automata that have the same number of states.

The minimization is based on the check the equivalence of two right languages of two states in a given DEA $A = (Q, \Sigma, \delta, s, f)$ for identifying indistinguishable states. For example in Fig. 5, we have to determine whether or not $L_{\overrightarrow{p}} = L_{\overrightarrow{q}}$. Since the regular expression equivalence problem is PSPACE-complete [8], the complexity of identifying indistinguishable states is at least PSPACE-complete. On the other hand, once we have all pairs of indistinguishable states, then the merging step in the micro-operation takes $O(|Q|)$ worst-case time for each pair of indistinguishable states, where $|Q|$ is the number of states in A, since each state can have at most $O(|Q|)$ in-transitions and $O(|Q|)$ out-transitions.

6. Conclusions

We have formally defined expression automata and DEAs based on the notion of prefix-freeness. In addition, we have shown that DEA languages are prefix-free regular languages and, therefore, they are a proper subfamily of regular languages.

State elimination is a natural way to compute a regular expression from a given automaton that results in an automaton that we call an expression automaton. One interesting observation about state elimination is that different state removal sequences from the same automaton give rise to different regular expressions that define the same language; see Fig. 8 for example.



$E_1 = (aa+b)(a+cb)^*(cd+d)$ $E_2 = (aa+b)a^*c(ba^*c)^*(ba^*d+d)+(aa+b)a^*d$

Figure 8: Different removal sequences result in different regular expressions for the same language. E_1 is the output of the state elimination in $p \rightarrow r \rightarrow q$ order and E_2 is the output of the state elimination in $p \rightarrow q \rightarrow r$ order, where $L(E_1) = L(E_2)$.

If we choose a good removal sequence, then we end up with shorter regular expressions. On the other hand, we have to consider all possible removal sequences to obtain the shortest regular expression by state elimination, which is exponential. Therefore, it is an interesting problem to develop heuristics that give a shorter regular expression by state elimination.

Acknowledgment

We are deeply grateful to Byron Choi for his helpful comments and hearty encouragement.

References

1. J. Brzozowski and E. McCluskey, Jr. Signal flow graph techniques for sequential circuit state diagrams. *IEEE Transactions on Electronic Computers*, EC-12:67–76, 1963.
2. S. Eilenberg. *Automata, Languages, and Machines*, volume A. Academic Press, New York, NY, 1974.
3. D. Giammarresi and R. Montalbano. Deterministic generalized automata. *Theoretical Computer Science*, 215:191–208, 1999.
4. J. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, 189–196, New York, NY, 1971. Academic Press.
5. J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 2 edition, 1979.
6. E. Moore. Gedanken experiments on sequential machines. In C. Shannon and J. McCarthy, editors, *Automata Studies*, 129–153, Princeton, NJ, 1956. Princeton University Press.
7. D. Perrin. Finite automata. In J. van Leeuwen, editor, *Formal Models and Seman-*

tics, volume B of *Handbook of Theoretical Computer Science*, 1–57. The MIT Press, Cambridge, MA, 1990.

8. L. Stockmeyer and A. Meyer. Word problems requiring exponential time. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, 1–9, 1973.
9. D. Wood. *Theory of Computation.* John Wiley & Sons, Inc., New York, NY, 1987.