The Generalization of Generalized Automata: Expression Automata

Yo-Sub Han and Derick Wood

Department of Computer Science,
The Hong Kong University of Science and Technology, Hong Kong
{emmous, dwood}@cs.ust.hk

Abstract. We explore expression automata with respect to determinism, minimization and primeness. 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 define the minimization of deterministic expression automata. Lastly, we discuss prime prefix-free regular languages.

Note that we have omitted almost all proofs in this preliminary version.

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 [4] have studied **generalized automata** (introduced by Eilenberg [3]) 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 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, and introduce prime prefix-free regular languages in Section 6.

2 Preliminaries

Let Σ denote a finite alphabet of characters and Σ^* denote the set of all strings over Σ, where the elements of Σ^* are called strings or blocks. We call an element of Σ a character and an element of Σ^* a string. A language over Σ is a 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 $xw = y$ and we define x to be a **proper prefix** of y is $x \neq \lambda$ and $x \neq y$. Given a set X of strings over Σ, X is **prefix-free** if no string in X is proper 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. A string x over Σ is accepted by A if there is a labeled path from s to a state in F such that this path spells out the string x. Thus, the language $L(A)$ of a finite-state automaton A is the set of all strings that are spelled out by paths from s to a final state in F. Automata that do not have any **useless states;** that is states that do not appear on any path from the start state to some final state are called **trim** or **reduced** [3, 9].

Eilenberg [3] 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 [4] define 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 L 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 there is a trim deterministic finite-state automaton (DFA) A for L that has no out-transitions from any final state.*

3 Expression Automata

It is well known that regular expressions and (deterministic) finite-state automata have exactly the same expressive power [6, 12]. A finite-state automaton allows only a single character in a transition and a generalized automaton [3] 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].

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 need 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 $\delta{:}Q \times Q \to \mathcal{R}_\Sigma$ 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 include a proof sketch that every finite-state automaton can be converted into an equivalent expression automaton and conversely.

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.*

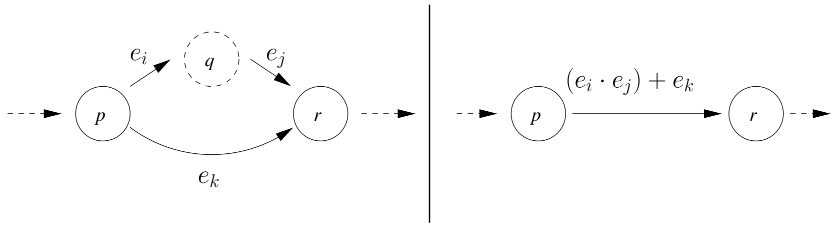

Fig. 1. An example of the state elimination of a state q

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 $q \in Q$. It is straightforward to show that any trim expression automaton A can be converted into a trim non-returning expression automaton for the same language $L(A)$.

We define the **state elimination** of $q \in Q \setminus \{s, f\}$ 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), $1 \leq i \leq m$, for some $m \geq 1$, for each out-transition (q, γ, r_j), $1 \leq j \leq n$, for some $n \geq 1$, 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 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.

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

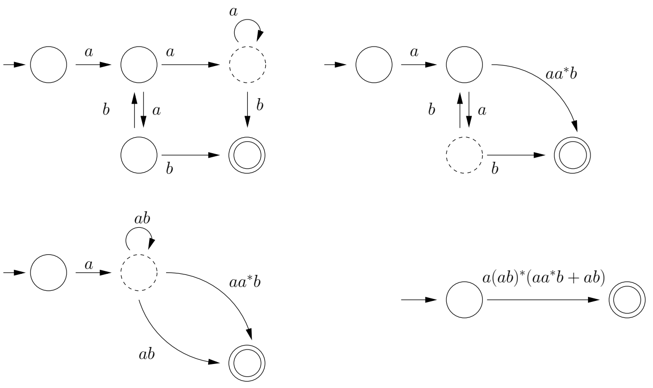

Fig. 2. An expression automaton for the regular language $L(a(ab)^*(aa^*b+ab))$ and its state eliminations

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.*

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 [12].

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 of the expression automaton in Figure 3(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 $1 \leq i \leq j$; hence, this expression automaton is not deterministic. On the other hand, the expression automaton in Figure 3(b) is deterministic since $L(a^*b)$ is a prefix-free language. We give a formal definition as following.

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:*

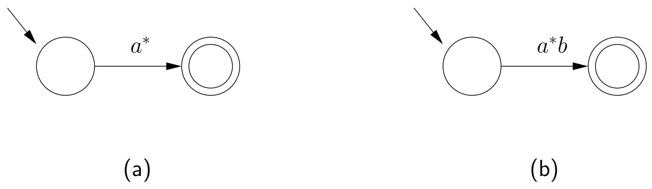

Fig. 3. a. Example of non-prefix-freeness. b. Example of prefix-freeness

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:** *For all $q \in Q$, $\delta(f, q) = \emptyset$.*

We use the acronym DEA to denote deterministic expression automaton.

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. However the converse does not hold.*

Proof. This result follows from Lemma 3 since the catenation of prefix-free languages is a prefix-free language.

Therefore, state elimination for a DEA preserves determinism.

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

Lemma 5 demonstrates that the regular languages accepted by DEAs are prefix-free and conversely. 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 to truly minimize DFAs [5, 8] in that they give unique (up to renaming of states) minimal DFAs. Recently, Giammarresi and Montalbano [4] suggested a minimization algorithm for **deterministic generalized automata (DGAs).** The technique does not however result in a unique minimal DGA. For a given 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 and one final states with one transition between them, from any DEA using a sequence of state eliminations.

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

We 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 exist a pair of indistinguishable states in the following states in a DFA. However, this property does not always hold for a DEA; see Fig. 4.

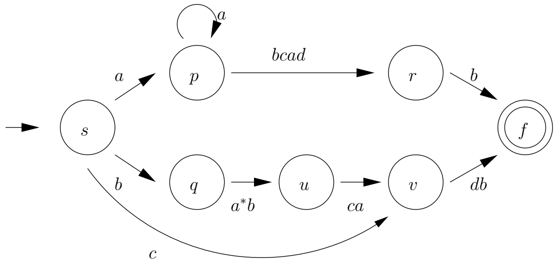

Fig. 4. An example of indistinguishable states. Note that r and u are distinguishable although $p \sim q$

Based on the notion of the right language, we define a minimal DEA as following.

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

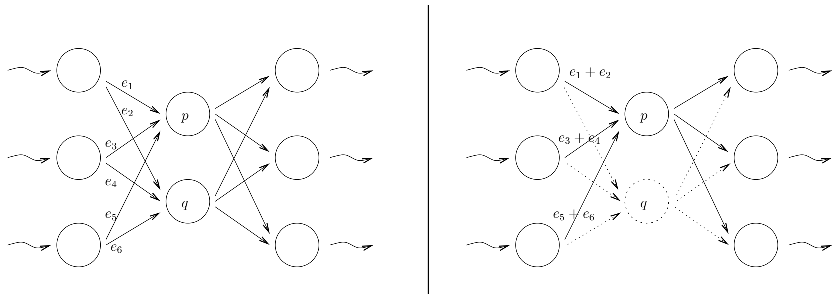

Fig. 5. An example of the merging two indistinguishable states p and q. The dotted lines show the removal of transitions

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.

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 language 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

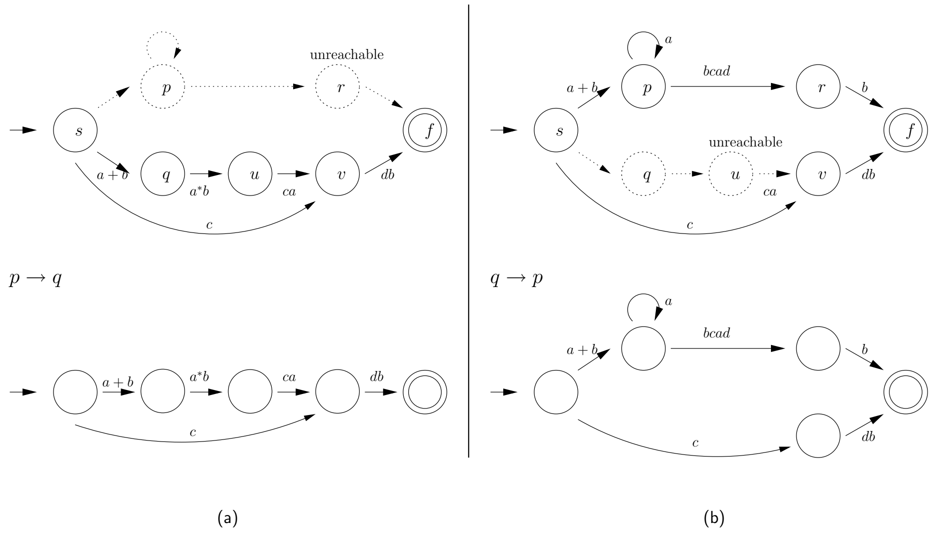

Fig. 6. Two different minimal DEAs for the DEA in Fig. 4

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.

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. 4 can be minimized in at least two different ways. As shown in Fig. 6(a), we merge p into q and remove state r which is now unreachable. In Fig. 6(b), we merge q into p and remove state u which is unreachable. But 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 numbers of states.

6 Prime Prefix-Free Regular Languages

Assume that we have the regular expressions $\alpha_1 = b^*a^*$ and $\alpha_2 = a^*b^*$. Once we catenate them however, $\alpha_1 \cdot \alpha_2 = b^*a^*b^*$ and we have only three stars, b^*, a^* and b^*, instead of four stars. Prefix-freeness ensures that there is no such loss as a result of catenation. Similarly, any infinite regular language, can be split unboundedly often. For example, $L(a^*) = L(a^*) \cdot L(a^*) \cdot L(a^*) \cdots L(a^*)$.

These two examples have led us to investigate whether an unbounded split is possible for an infinite prefix-free regular language. There are some known results on the prime decomposition of finite languages and decomposition of regular languages [7, 10].

Definition 6. *A prefix-free regular language L is* **prime** *if $L \neq L_1 \cdot L_2$ for any two non-trivial prefix-free regular languages L_1 and L_2.*

We say a state b in a DFA A is a **bridge state** if the following conditions hold:

1. b is neither a start nor a final state.
2. For any string $w \in L(A)$, its path in A must pass through b at least once.

Then we partition A at b into two subautomata A_1 and A_2 such that all out-transitions from b belong to A_2 and make b to be the final state of A_1 and the start state of A_2, respectively. It ensures that A_1 defines a prefix-free regular language.

Theorem 3. *A prefix-free regular language L is a prime prefix-free regular language if and only if there is no bridge state in the minimal DFA A for L.*

Theorem 3 shows that a given prefix-free regular language L cannot be split unboundedly often because its minimal DFA has a finite number of states.

7 Conclusion

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. 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.

We have studied the minimization of DEA and demonstrated that minimization is not unique in general. Since the regular expression equivalence problem is PSPACE-complete [11], we believe that the complexity of minimization is at least PSPACE-complete.

Acknowledgments

We are deeply grateful to Byron Choi for his helpful comments and hearty encouragement. We appreciate Wojciech Fraczak for providing an example to solidify the definition of bridge states in Section 6. We also discovered that the prime prefix-free regular language decomposition has been studied already by Czyzowicz et al. [2] while preparing the final version of this paper for the proceedings.

The authors were supported under the Research Grants Council Competitive Earmarked Research Grant HKUST6197/01E.

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. J. Czyzowicz, W. Fraczak, A. Pelc, and W. Rytter. Linear-time prime decomposition of regular prefix codes. *International Journal of Foundations of Computer Science*, 14:1019–1032, 2003.
3. S. Eilenberg. *Automata, Languages, and Machines*, volume A. Academic Press, New York, NY, 1974.
4. D. Giammarresi and R. Montalbano. Deterministic generalized automata. *Theoretical Comput. Sci.*, 215:191–208, 1999.
5. 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*, pages 189–196, New York, NY, 1971. Academic Press.
6. J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 2 edition, 1979.
7. A. Mateescu, A. Salomaa, and S. Yu. Factorizations of languages and commutativity conditions. *Acta Cybernetica*, 15(3):339–351, 2002.
8. E. Moore. Gedanken experiments on sequential machines. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 129–153, Princeton, NJ, 1956. Princeton University Press.

9. D. Perrin. Finite automata. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 1–57. The MIT Press, Cambridge, MA, 1990.
10. A. Salomaa and S. Yu. On the decomposition of finite languages. In G. Rozenberg and W. Thomas, editors, *Developments in Language Theory (DLT) 99*, pages 22–31. World Scientific, 2000.
11. L. Stockmeyer and A. Meyer. Word problems requiring exponential time. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, pages 1–9, 1973.
12. D. Wood. *Theory of Computation*. John Wiley & Sons, Inc., New York, NY, 1987.