Gluskov and Thompson Constructions: A Synthesis*

Dora Giammarresi† Jean-Luc Ponty‡ Derick Wood§

August 5, 1998

Abstract

We reexamine the relationship between the two most popular methods for transforming a regular expression into a finite-state machine: the Glushkov and Thompson constructions. These methods have received more attention recently because of the Standard Generalized Markup Language (SGML) and a revival of interest in symbolic toolkits for regular and context-free expressions, grammars, and machines. We establish that:

- Every Thompson machine is, in a sense we make precise, a Glushkov machine
- Every Glushkov machine is hidden in the corresponding Thompson machine

In addition, we establish that a number of other inductive constructions yield machines that have their corresponding Glushkov machine hidden in them.

1 Introduction

The fundamental and seminal results of Kleene [19] included a solution for the synthesis problem for finite-state machines from regular expressions. He gave an inductive construction of an equivalent finite-state machine from a regular expression. The approach is a popular one no doubt because regular expressions are defined inductively; indeed, textbooks on the theory of computation always (as far as we are aware) use inductive construction; for example, the compiler-design text of Aho and his coauthors [1], the compiler-theory texts of Aho and Ullman [2] and of Sippu and Soisalon-Soininen [24], the formal-languages text of Hopcroft and Ullman [16], and the theory of computation texts of Denning and his coauthors [11], Drobot [12], Lewis and Papadimitriou [20], Martin [21], and Wood [27]. Moreover,

*This research was supported under a grant from the Research Grants Council of Hong Kong SAR. It was carried out while the first and second authors were visiting HKUST.

†Dipartimento di Matematica Applicata e Informatica, Università Ca' Foscari di Venezia, via Torino 155, 30173 Venezia Mestre, Italy.

‡L.I.F.A.R., Université de Rouen, Faculté des Sciences et des Techniques, Place Émile Blondel, 76821 Mont-Saint-Aignan Cedex, France.

§Department of Computer Science, Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, Hong Kong SAR.

inductive constructions are a popular implementation technique in system utilities, such as `awk` and `grep`, and software tools for regular expression manipulation such as AMoRE [18], AUTOMATE [9] and Grail [23]. When implementing regular-expression tools, regular expressions are usually, but not always, compiled into an executable form based on finite-state machines. Thus, the size of a finite-state machine and the time taken to construct it are important measures of efficiency. In addition, the simplicity of a machine is also a criterion in some situations since it can lead to simple representations. One well-known example that has these attributes is the Thompson inductive construction [25]. It produces finite-state machines that have size linear in the size of the given expressions and does so in linear time. Crucually, it also gives machines that have at most two transitions into and at most two transitions out of each each state—their structure is simple. It was used in the first implementation of `grep`.

A second construction that is older than Thompson's is the Glushkov construction [14, 15]. It is a two-step construction. First, inductively compute three sets of symbols and then, second, compute the equivalent machine directly from these sets. About 10 years ago, Champarnaud [8] observed that the inductive construction they used in AUTOMATE [7, 9] gave machines that looked like ones given by the Glushkov construction. The proof of this identity was first sketched by him [8] and, more recently, proved rigorously by Champarnaud and his coresearchers [10]. There has been a revival of interest in the Glushkov construction since it can be used to characterize those regular expressions used in the Standard Generalized Markup Language (SGML) [17] that are unambiguous according to the SGML standard [5]. They are exactly those expressions for which the Glushkov construction gives deterministic machines. Berry and Sethi [3] also argue that the Glushkov construction gives natural machines.

Watson [26] presents, in his thesis, a detailed taxonomy of the constructions of finite-state machines from regular expressions. He relates the various constructions to his variant of the Thompson construction which is based on dotted regular expressions. As such his interest is different from ours but complementary to it.

We contribute to the explication of the relationship of the Thompson and Glushkov constructions in two ways. First, we prove, in a way we make precise, that we can use the Glushkov construction to construct the machines obtained by the Thompson construction; see Section 4.1. We preprocess the given regular expressions and postprocess the resulting machines.

Second, we can use the Thompson construction to construct machines obtained by the Glushkov construction; see Section 4.2. We postprocess the resulting machines.

In addition, we also argue that most of the inductive constructions found in the literature yield machines that can be converted into Glushkov machines using the same technique; see Section 5.

2 Notation and terminology

We first review of the definition of a regular expression before defining finite-state machines. Let Σ be an alphabet. Then, we inductively define a regular expression E over Σ as follows: First, we have three base cases:

$E = \emptyset$, where $\emptyset$ denotes the empty set;

$E = \lambda$, where λ denotes the null string;

$E = a$, where a is in Σ.

Second, we have four inductive cases:

$E = (F + G)$, where F and G are regular expressions;

$E = (F \cdot G)$, where F and G are regular expressions;

$E = (F^*)$, where F is a regular expression;

$E = (F)$, where F is a regular expression.

We define the language of E inductively, in the usual way [2, 28]. To be precise, for a regular expression E, $L(E)$ is defined inductively as follows:

$L(E) = \emptyset$, if $E = \emptyset$;

$L(E) = \{\lambda\}$, if $E = \lambda$;

$L(E) = \{a\}$, if $E = a$;

$L(E) = L(F) \cup L(G)$, if $E = (F + G)$;

$L(E) = L(F)L(G)$, if $E = (F \cdot G)$;

$L(E) = L(F)^*$, if $E = (F^*)$;

$L(E) = L(F)$, if $E = (F)$.

The **size** $|E|$ of a regular expression is defined to be the total number of appearances of operators and symbols in E (we include the null-string and empty-set symbols in the count). An expression E is **empty-free** if E does not contain $\emptyset$ as a subexpression.

We next recall the definition of finite-state machines. A **finite-state machine M** consists of a finite set Q of states, an input alphabet Σ, a start state $s \in Q$, a set $F \subseteq Q$ of final states and a transition relation $\delta \subseteq Q \times \Sigma_\lambda \times Q$. Clearly, we can depict the transition relation of such a machines as an edge-labeled digraph (the labels are either symbols from Σ or the null string); it is usually called the state digraph or transition digraph of the machine. If we drop the edge labels (we consider the null string to be a label for this purpose) of a state digraph and ignore multiple edges, we obtain a digraph, the **underlying digraph** of the machine. An **accepting computation** of a machine M on a string $x \in \Sigma^*$ is a path from the start state to some final state that spells out the string x. The **language of a machine M** is the set of all strings that have an accepting computation in M.

The **size** $|M|$ of a finite-state machine is defined to be the the total number of transitions in M.

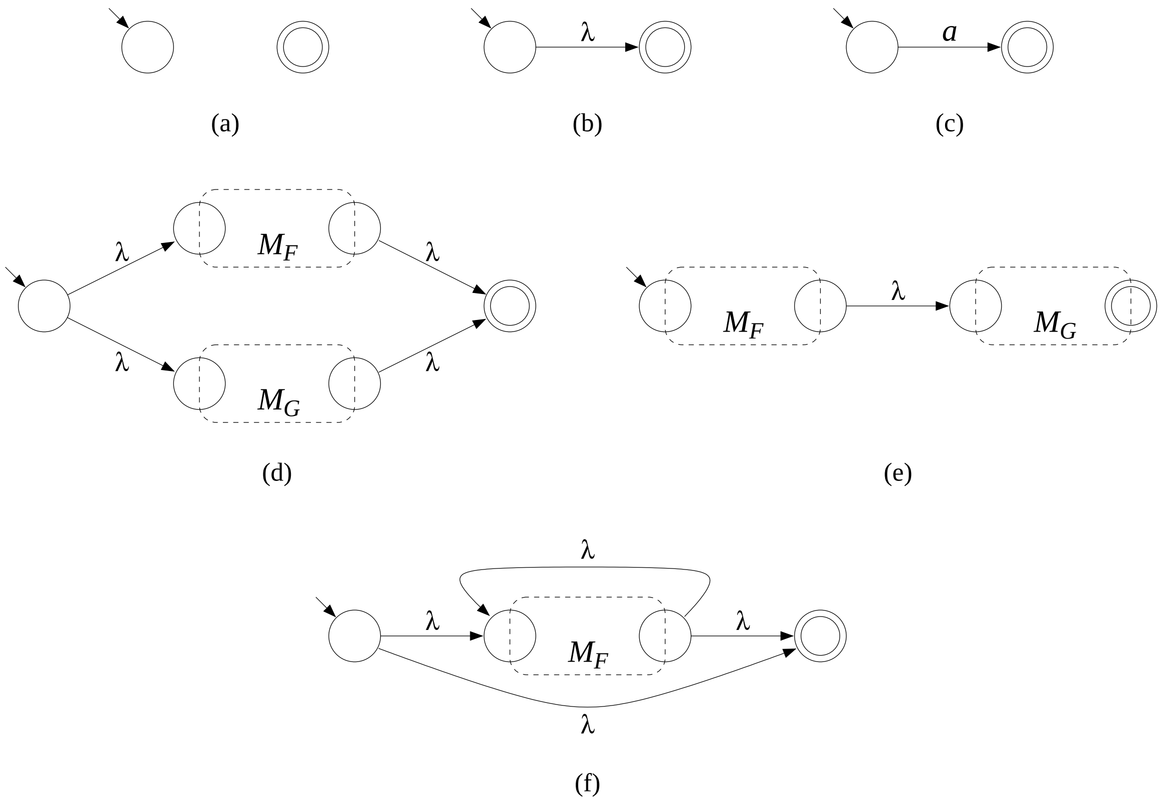

Figure 1: The Thompson construction. The order of the figures corresponds to the order of the cases in the definition of regular expressions. The finite-state machines correspond to the regular expressions: a. $E = \emptyset$; b. $E = \lambda$; c. $E = a$, $a \in \Sigma$; d. $E = (F + G)$; e. $E = (F \cdot G)$; and f. $E = (F^*)$.

3 Regular expressions to finite-state machines

We first recall the two most popular constructions that produce finite-state machines from regular expressions: the Thompson and Glushkov constructions. Then, we then provide an inductive version of the Glushkov construction that was designed by Champarnaud [8, 10].

In all cases, we require that a machine M obtained from a regular expression E satisfies the condition that $L(M) = L(E)$; that is, M and E are equivalent.

3.1 The Thompson construction

Thompson designed this construction in 1968 [25] to compile regular expressions into a form that is suitable for text searching. The construction is defined inductively (see Fig. 1) and it gives a finite-state machine that has a number of pleasant properties:

1. There is one start state that is **only exiting;** that is, there are no transitions that enter it.

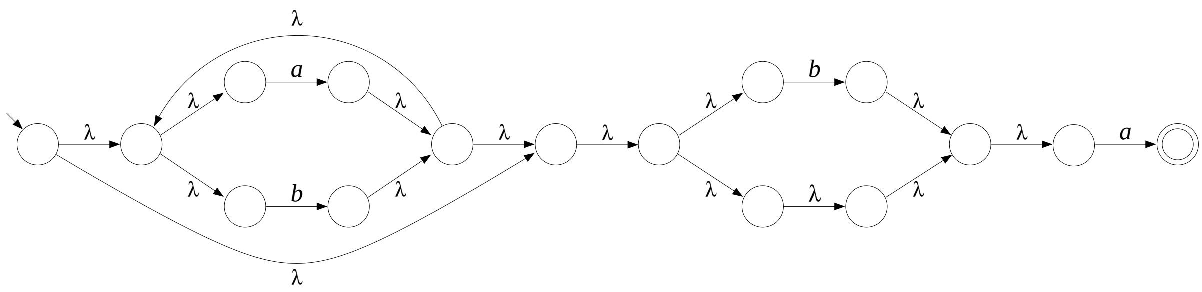

Figure 2: The result of the Thompson construction on the running example expression $(((a+b)^*)\cdot((b+\lambda)\cdot a))$.

2. There is one final state that is **only entering;** that is, there are no transitions that leave it.

3. Each state has at most two transitions leaving it and at most two transitions entering it.

4. The size of the machine is at most three times the size of the given regular expression.

Given a regular expression E, we denote the machine obtained by the Thompson construction by M_E^T; we call M_E^T a **Thompson machine.** We focus attention on Thompson machines for empty-free expressions; we call them **empty-free Thompson machines.** In Fig. 2, we give the result of the Thompson construction on the regular expression $(((a+b)^*)\cdot((b+\lambda)\cdot a))$ which we use as a running example throughout. As a final remark about Thompson machines, observe that the reverse of a Thompson machine (reverse all its transitions, make the start state final and the final state start) is also a Thompson machine. The construction is symmetric.

3.2 The Glushkov construction

Glushkov first suggested this construction in 1960 [14, 15]; it was also suggested by McNaughton and Yamada [22] independently and at about the same time. The original construction is based on the first, last and follow sets of positions in the given regular expression. The appearances, from left to right, of the Σ-symbols in a regular expression are numbered from one to the total number of appearances. The appearances are called **positions** and we modify a regular expression E to obtain a new regular expression E' in which each symbol is replaced by its position; thus, if there are n appearances, E' is a regular expression over $\{1,\ldots,n\}$. If $a\in\Sigma$ is at a position i, then we say that i **corresponds** to a. We now define the three sets of positions as follows:

first(E) is the set of all positions that can begin a string in $L(E')$.

last(E) is the set of all positions that can end a string in $L(E')$.

follow(i,E) is the set of all positions in $L(E')$ that can follow position i.

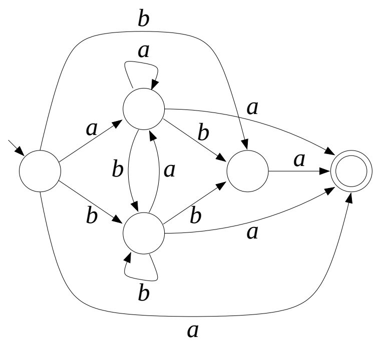

Figure 3: An example of Glushkov's construction on the running example.

Once we have computed these sets, we can construct a finite-state machine M_E directly as follows: *The states of* M_E *are* $\{0, \ldots, n\}$, 0 *is the start state, last*(E) *is the set of final states, and the transitions are*

$$\{(i, a, j) : j \text{ corresponds to } a \text{ and } j \in \mathit{follow}(i, E) \text{ or } i = 0 \text{ and } j \in \mathit{first}(E)\}.$$

In Fig. 3, we display the result of the construction on the running example expression $(((a + b)^*) \cdot ((b + \lambda) \cdot a))$. The **Glushkov machine,** as we call it, also has a number of interesting properties:

1. There is one start state that is only exiting.
2. It has no null transitions.
3. For each state p, all transitions into p have the same label.
4. The size of the Glushkov machine of a regular expression E is, in the worst case, $O(|E|^2)$.

3.3 The Champarnaud–Glushkov construction

Champarnaud [8], and Champarnaud and his associated researchers [10] proved that the Glushkov construction can be made inductive in the same style as the Thompson construction. The inductive construction (see Fig. 4) is similar to, but different from, the Thompson construction (see Fig. 1). The base cases in Fig. 4 should be clear. We give more explanation of the inductive cases. The construction of the machine M_{F+G} from M_F and M_G combines the two machines by identifying (or merging) their start states.

The inductive step for $F \cdot G$ identifies the start state of M_G separately with each final state of M_F. The effect of the identification is that each, previously final, state in M_F has transitions to all states that M_G's start state has transitions to and, also, M_G's start state disappears. If M_G's start state was a final state, then M_F's final states remain final; otherwise, the become nonfinal.

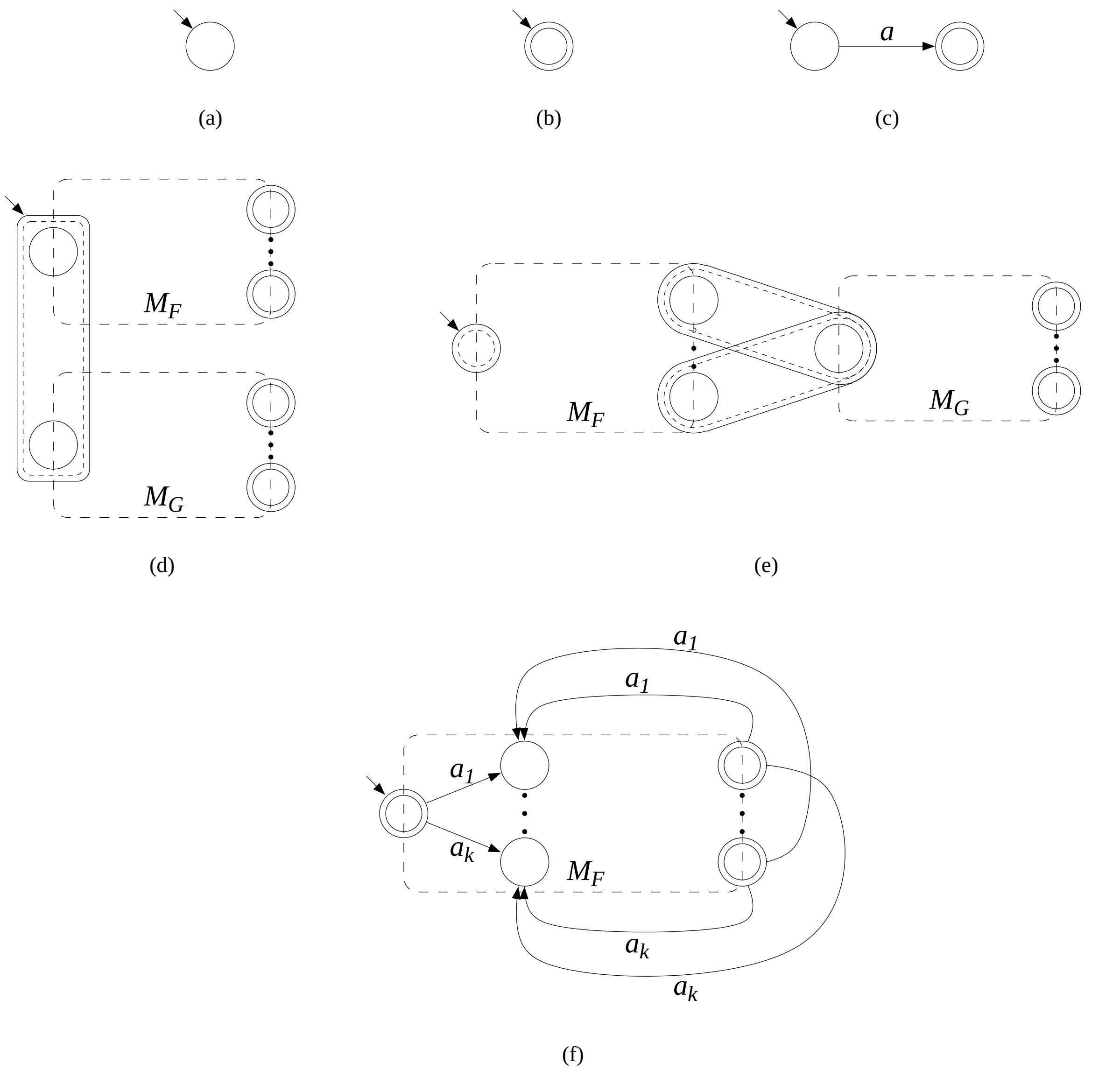

Figure 4: The Champarnaud inductive construction of the Glushkov machine. The finite-state machines correspond to the regular expressions: a. $E = \emptyset$; b. $E = \lambda$; c. $E = a$, $a \in \Sigma$; d. $E = (F + G)$; e. $E = (F \cdot G)$; and f. $E = (F^*)$.

The inductive step for F^* is similar to that for product. It identifies the start state of M_F separately with each final state of M_F. The difference is that the start state does not disappear, it becomes a final state and all other final states remain as final states. The effect of the identification is that each final state in M_F is given transitions to all states that M_F's start state has transitions to.

4 The Glushkov–Thompson relationship

There are a number of issues raised by the Glushkov and Thompson constructions. The most basic is: *How are the Glushkov and Thompson machines related?* We explore their relationship from a new viewpoint. We first demonstrate that every Thompson machine is a disguised Glushkov machine in Section 4.1. Then, in Section 4.2, we prove that each Thompson machine can be transformed into an equivalent Glushkov machine using a specific null-elimination transformation; that is, every Glushkov machine is a disguised Thompson machine.

4.1 The Glushkov construction of a Thompson machine

Let τ be a new symbol that is not in Σ. Now, define the **τ-expansion of a regular expression E** over Σ to be a regular expression E_τ over $\Sigma \cup \{\tau\}$ that is defined inductively as follows:

$E_\tau = \emptyset$ if $E = \emptyset$;

$E_\tau = \tau$ if $E = \lambda$;

$E_\tau = a$ if $E = a$;

$E_\tau = (((\tau \cdot F_\tau) + (\tau \cdot G_\tau)) \cdot \tau)$ if $E = (F + G)$;

$E_\tau = ((F_\tau \cdot \tau) \cdot G_\tau)$ if $E = (F \cdot G)$;

$E_\tau = (((\tau \cdot F_\tau)^*) \cdot \tau)$ if $E = (F^*)$;

$E_\tau = (F_\tau)$ if $E = (F)$.

When we apply the Glushkov construction to an empty-free expression E_τ, the machine M_τ that we obtain gives the Thompson machine for E if we replace all appearances of τ with the null string. We apply the τ-expansion to our running example and obtain the machine shown in Fig. 5.

Define a projection morphism $h_\tau : (\Sigma \cup \{\tau\})^* \longrightarrow \Sigma^*$ by: $h_\tau(a) = a$, for all $a \in \Sigma$, and $h_\tau(\tau) = \lambda$ and extend it to be machine morphism by: For a finite-state machine $M = (Q, \Sigma \cup \{\tau\}, \delta, s, F)$, $h_\tau(M) = (Q, \Sigma, h_\tau(\delta), s, F)$, where $h_\tau((p, a, q)) = (p, h_\tau(a), q)$, for all $(p, a, q) \in \delta$.

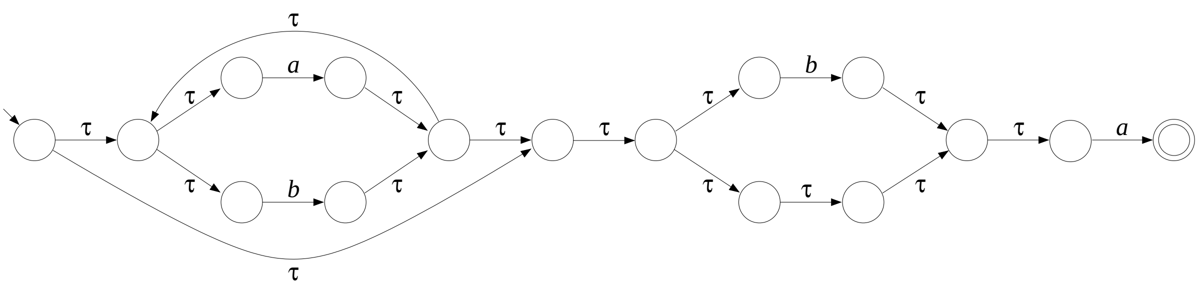

Figure 5: The Glushkov machine obtained from the τ-expansion of the running example expression.

Theorem 1 *Let E be an empty-free regular expression over Σ, and let E_τ be its τ-expanded form, where $\tau \notin \Sigma$. Then, the Thompson machine for E is identical, up to a renaming of states, to the Glushkov machine for E_τ when we replace all occurrences of τ with the null string; that is,*

$$h_\tau(M^G_{E_\tau}) \cong M^T_E.$$

Proof: We prove this result by induction using the Champarnaud–Glushkov and the Thompson inductive constructions defined in Figs. 4 and 1. We induct on the size of E. If $|E| = 1$, then either $E = \lambda$ or $E = a$ in which case $E_\tau = \tau$ or $E_\tau = a$, respectively. Clearly, $h_\tau(M^G_{E_\tau}) \cong M^T_E$ in both cases.

Now, assume that the result holds for all expressions E with $|E| \leq n$, for some $n \geq 1$, and consider an expression E of size $n+1$. Since $n+1 \geq 2$, E is not one of the base cases. Indeed, $E = (F + G)$, $E = (F \cdot G)$, or $E = (F^*)$. Although we examine each subcase separately, we can assume, by the induction hypothesis, that $h_\tau(M^G_{F_\tau}) \cong M^T_F$ and $h_\tau(M^G_{G_\tau}) \cong M^T_G$.

$E = (F + G)$**.** By definition, the τ-expansion of E is $E_\tau = (((\tau \cdot F_\tau) + (\tau \cdot G_\tau)) \cdot \tau)$. The corresponding Glushkov machine is given in Fig. 6(d). Clearly, by comparing Figs. 1(d) and 6(d), $h_\tau(M^G_{E_\tau}) \cong M^T_E$ in this case.

$E = (F \cdot G)$**.** By definition, the τ-expansion of E is $E_\tau = ((F_\tau \cdot \tau) \cdot G_\tau)$. Clearly, by comparing Figs. 1(e) and 6(e), $h_\tau(M^G_{E_\tau}) \cong M^T_E$ in this case.

$E = (F^*)$**.** By definition, the τ-expansion of E is $E_\tau = (((\tau \cdot F_\tau)^*) \cdot \tau)$. Clearly, by comparing Figs. 1(f) and 6(f), $h_\tau(M^G_{E_\tau}) \cong M^T_E$ in this case.

$E = (F)$**.** By definition, the τ-expansion of E is $E_\tau = (F_\tau)$.

$\square$

There are three interesting implications of Theorem 1. First, it leads to a two-step construction for the Thompson machine, rather the usual inductive construction. We can use the two-step Glushkov construction on the τ-expanded version E_τ of an empty-free regular expression E to give the Glushkov machine for E_τ. We can then apply h_τ to this machine to obtain the Thompson machine for E.

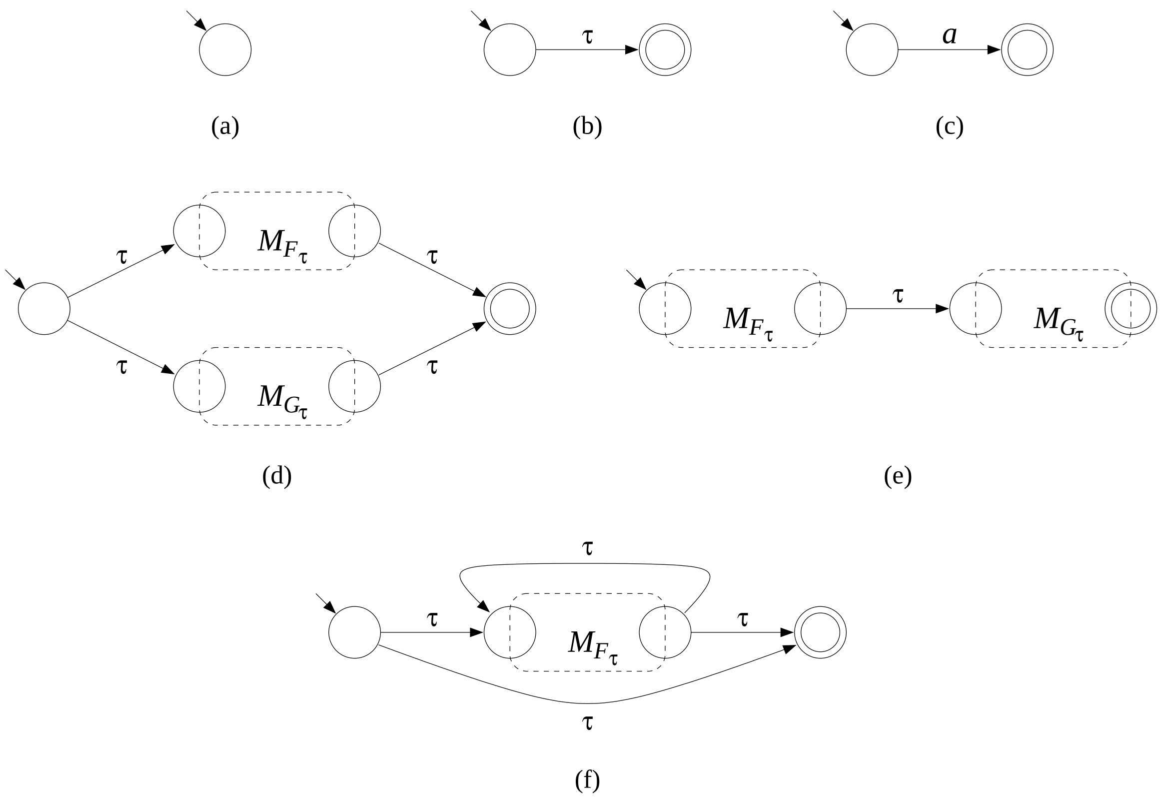

Figure 6: The Champarnaud–Glushkov induction construction for τ-expanded regular expressions as used in the proof of Theorem 1. The finite-state machines correspond to the regular expressions: a. $E = \emptyset$; b. $E = \lambda$; c. $E = a$, $a \in \Sigma$; d. $E = (F + G)$; e. $E = (F \cdot G)$; and f. $E = (F^*)$.

Corollary 2 *Let E be an empty-free regular expression over Σ, and let E_τ be its τ-expanded form, where $\tau \notin \Sigma$. Then, the Thompson machine for E can be obtained using the first, last and follow sets of E_τ to construct the Glushkov machine for E_τ directly. We then apply h_τ to the resulting machine.*

Second, we have proved that every Thompson machine obtained from an empty-free expression is also a Glushkov machine if we interpret the null-string symbol as a symbol of the alphabet. An alternative viewpoint is captured in the following result.

Corollary 3 *The underlying digraph of the Thompson machine of an empty-free expression is the underlying digraph of a Glushkov machine.*

We have characterized [13] the underlying digraphs of Thompson machines and Caron and Ziadi [6] have characterized the underlying digraphs of Glushkov machines.

Third, we can remove the empty-free restriction on regular expressions in Theorem 1 if we reduce the corresponding Glushkov and Thompson machines. Recall that a machine is **reduced** if either every state and every symbol is used in some accepting computation, or the machine has only one state, the start state, which is not final, and an empty transition relation.

Corollary 4 *Let E be a regular expression over Σ, and let E_τ be its τ-expanded form, where $\tau \notin \Sigma$. Then, the reduced Thompson machine for E is identical, up to a renaming of states, to the reduced Glushkov machine for E_τ when we replace all occurrences of τ with the null string; that is,*

$$h_\tau(\mathit{reduce}(M^G_{E_\tau})) \cong \mathit{reduce}(M^T_E).$$

4.2 The conversion of a Thompson machine into a Glushkov machine

The second result for the two constructions is that every Glushkov machine is hidden in the corresponding Thompson machine. We introduce a null-elimination transformation that removes null transitions and preserves some specific properties of the machines. When applied repeatedly to a Thompson machine to remove all null transitions, it yields a corresponding Glushkov machine. Brüggemann-Klein and Wood [4] were the first to claim such a result. We provide a rigorous proof of this fact and also establish some specific properties of the transformation. Sippu and Soisalon-Soininen [24] introduced a similar transformation to remove null transitions in a more general setting.

The specific null-elimination transformation is illustrated in Fig. 7. We denote by $\mathrm{elim}(M, q)$ the finite-state machine that is obtained from a finite-state machine $M = (Q, \Sigma, \delta, s, F)$ by applying null elimination to remove the null transitions into state q from machine M. In most circumstances it also removes all transitions out of q and q itself as well. We formally define $\mathrm{elim}(M, q)$ as follows: First, the state q must satisfy the **null-elimination precondition:** *All transitions into state q are null transitions and there is at least one such transition.* Second, if q has a null self-loop, we remove it. Third, let there be k null transitions into q, where $k \geq 1$, and m transitions out of q, where $m \geq 0$. Let (p_i, λ, q), $1 \leq i \leq k$, be the

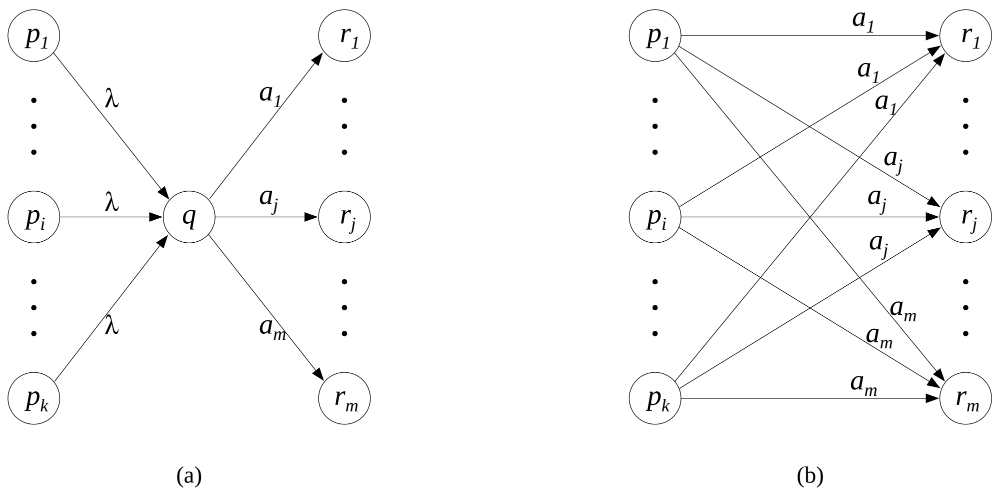

Figure 7: The null-elimination transformation used to convert a Thompson machine into a Glushkov machine.

transitions into q and (q, a_j, r_j), $0 \le j \le m$, be the transitions out of q. Then, we define $\mathrm{elim}(M, q)$ to be the machine $(Q', \Sigma, \delta', s, F')$, where

$$
\begin{aligned}
Q' &= \begin{cases} Q, & \text{if } q = s, \\ Q - \{q\}, & \text{otherwise;} \end{cases} \\
\bar{\delta} &= \delta - \{(p, \lambda, q) : 1 \le i \le k\}, \\
&\quad \cup \{(p_i, a_j, r_j) : 1 \le i \le k \text{ and } 0 \le j \le m\}; \\
\delta' &= \begin{cases} \bar{\delta}, & \text{if } q = s, \\ \bar{\delta} - \{(q, a_j, r_j) : 0 \le j \le m\}, & \text{otherwise;} \end{cases} \\
\bar{F} &= F \cup \{p_i : 1 \le i \le k\}; \\
F' &= \begin{cases} \bar{F}, & \text{if } q \text{ is final and start,} \\ \bar{F} - \{q\}, & \text{if } q \text{ is final and nonstart,} \\ F, & \text{otherwise.} \end{cases}
\end{aligned}
$$

Note that null elimination also removes state q except when q is the start state. Observe that when q has no outgoing transitions, the transformation removes q and all null transitions into q. In this case, if q is a final state, then the states p_i become final states.

We first establish some fundamental properties of the transformation before proving that it can be used to obtain a Glushkov machine from a Thompson machine. A finite-state machine M is a **homogeneous** if, for all states p in M, all transitions into p have the same label. We treat the null string as a label in this case. We also write that a **state is homogeneous** if all transitions into it have the same label. Observe that all Glushkov and Thompson machines are homogeneous. We have the following homogeneity preservation property.

Lemma 5 *Let M be a finite-state machine and q be a state in M that satisfies the null-*

elimination precondition. Then, M is homogeneous if and only if $\mathrm{elim}(M, q)$ *is homogeneous.*

Proof: We use the notation in the definition of $\mathrm{elim}(M, q)$. First note that the only difference between $\mathrm{elim}(M, q)$ and and M is that $\mathrm{elim}(M, q)$ has no null transitions into q but it has new transitions that connect the p_i's to the r_j's. Also $\mathrm{elim}(M, q)$ may not have the state q and its transitions to the r_j's.

If $\mathrm{elim}(M, q)$ is homogeneous, then we need to argue that the r_j's and q are homogeneous in M. By definition, q is homogeneous in M. Moreover, each r_j is homogeneous in M since it is homogeneous in $\mathrm{elim}(M, q)$ and removing in transitions of r_j in $\mathrm{elim}(M, q)$ does not change the homogeneity of r_j.

Conversely, if M is homogeneous, then removing the in transitions of q does not change its homogeneity. In addition, replacing those in transitions with new transitions from the p_i's to the r_j's does not change the homogeneity of the r_j's since we add only transitions with the same label a_j into each r_j. Thus, $\mathrm{elim}(M, q)$ is homogeneous. $\square$

A second preservation result concerns the underlying digraph of a finite-state machine. Recall that a **digraph** G consists of a finite set V of vertices and a set E of directed edges of the form (u, v), where u and v are vertices. A **path** is a sequence $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ of edges; it is a **cycle** if $v_0 = v_k$ and $k \geq 1$. A path is a **simple path** if it contains no cycles.

We are particularly interested in digraphs that have a single designated **source vertex** that has no edges entering it, one or more designated **sink vertices,** and such that each of its vertices is on some simple path from the source to a sink. Such digraphs are called **hammocks**[1]. Note that the underlying digraphs of Glushkov machines and of empty-free Thompson machines are hammocks.

Lemma 6 *If M is a finite-state machine whose underlying digraph is a hammock and there is a state q in M that satisfies the null-elimination precondition, then the underlying digraph of* $\mathrm{elim}(M, q)$ *is also a hammock.*

Proof: Since M is a hammock, the states p_i, q and r_j are on simple paths from the source vertex to some sink vertices. If q is sink vertex, then the p_i become sink vertices.

Any simple source-sink path for a vertex r_j in M that passes through q must use an edge (p_i, q) to reach q; therefore, after null elimination, there is a simple path that will use the edge (p_i, r_j) to reach r_j.

Similarly, any simple source-sink path for a vertex p_i in M that passes through q before it passes through p_i must use an edge (p_t, q) to reach q and also follow an edge (q, r_j). Therefore, after null elimination, we can replace the subpath $(p_t, q), (q, r_j)$ with the edge (p_t, r_j) to obtain a simple source-sink path in $\mathrm{elim}(M, q)$ that includes p_i. $\square$

We now establish that null elimination is confluent; that is, the order of application of null-elimination transformations to a specific finite-state machine does not matter, all permutations of their order yield the same machine.

[1]In the literature hammocks are usually called *st*-digraphs. Also hammocks are usually defined to have a single sink that has no edges exiting from it; however, we can always add a universal sink node to a hammock in our sense to ensure that it is a hammock in the usual sense.

Lemma 7 *For all finite-state machines M with two distinct states q_1 and q_2 that satisfy the null-elimination precondition, we have*

$$\mathrm{elim}(\mathrm{elim}(M, q_1), q_2) = \mathrm{elim}(\mathrm{elim}(M, q_2), q_1).$$

Proof: Let $\bar{Q_1} = \{p_{1,1}, \ldots, p_{1,k_1}, q_1, r_{1,1}, \ldots, r_{1,m_1}\}$ and $\bar{Q_2} = \{p_{2,1}, \ldots, p_{2,k_2}, q_2, r_{2,1}, \ldots, r_{2,m_2}\}$. Now, if $\bar{Q_1} \cap \bar{Q_2} = \emptyset$, then the two null eliminations do not interact at all and the claim holds.

We now consider the the interacting cases; that is, $\bar{Q_1} \cap \bar{Q_2} \neq \emptyset$. First, we have the boundary cases for which $\{q_1, q_2\} \cap \bar{Q_1} \cap \bar{Q_2} = \emptyset$. In all these cases, the order of application of the two null eliminations is immaterial. We give two cases to illustrate the argument. If there is a state p that has transitions (p, λ, q_1) and (p, λ, q_2), then the null eliminations will give p new transitions $(p, a_{1,j}, r_{1,j})$, for all j, $1 \leq j \leq m_1$, and $(p, a_{2,j}, r_{2,j})$, for all j, $1 \leq j \leq m_2$, independently of the order of null elimination. Also, if there is a state r that has an in transition (q_1, a, r) and an out transition (r, λ, q_2), then the null eliminations will give r new transitions $(p_{1,i}, a, r)$, for all i, $1 \leq i \leq k_1$, and $(r, a_{2,j}, r_{2,j})$, for all j, $1 \leq j \leq m_2$, independently of the order of null elimination.

Second, we have the cases for which $\{q_1, q_2\} \cap \bar{Q_1} \cap \bar{Q_2} \neq \emptyset$. We again consider one example case. Assume without loss of generality that $q_1 = r_{2,1}$. Because q_1 has only null in transitions, we infer that $a_{2,1} = \lambda$, (q_2, λ, q_1) is in M and $q_2 = p_{1,i}$, for some i, $1 \leq i \leq m_1$. Without loss of generality, assume that $i = 1$. Now, computing the transitions we obtain from these facts, if we apply null elimination to q_1 first, then we obtain $(p_{1,1} = q_2, a_{1,j}, r_{1,j})$, for all j, $1 \leq j \leq m_1$. After applying null elimination to q_2, we then obtain $(p_{2,i}, a_{2,j}, r_{2,j})$, for all j, $2 \leq j \leq m_2$, and for all i, $1 \leq i \leq k_2$, and $(p_{2,i}, a_{1,j}, r_{1,j})$, for all j, $1 \leq j \leq m_1$. for all i, $1 \leq i \leq k_2$. If we do the null elimination of q_2 first, then we obtain the same transitions for the $p_{2,i}$. □

We can now state and prove the main result of this section.

Theorem 8 *For each empty-free regular expression E, if M_E^T has exactly k distinct states $q_1, \ldots, q_k$ that satisfy the null-elimination precondition, then*

$$\mathrm{elim}(\cdots \mathrm{elim}(M_E^T, q_1), \ldots, q_k) \cong M_E^G.$$

Proof: We prove the result by inductively constructing a set of null-elimination transformations on a machine M_E^T that yields a finite-state machine without null transitions. In addition, the application of null elimination at each induction step of the Thompson construction (see Fig. 1) yields the corresponding step in the inductive construction of a Glushkov machine; see Fig. 4. By Lemma 7, we can then conclude the result holds.

The base cases in Figs. 1(b) and (c) yield, under null elimination, the cases in Figs. 4(b) and (c), respectively. Now, consider the three induction cases: $F + G$, $F \cdot G$ and F^*. Given their Thompson constructions in Figs. 1(d),(e) and (f), null elimination yields the Champarnaud–Glushkov constructions in Figs. 4(d), (e) and (f). Note that M_{F+G}^T requires three applications of null elimination, $M_{F \cdot G}^T$ one application and $M_{F^*}^T$ two applications. We have completed the induction step; thus, the theorem holds. □

5 Other inductive constructions

There are a surprising variety of inductive constructions in the literature. We examine some of them. We single out the construction of Sippu and Soisalon-Soininen [24], in Section 5.1, since they designed a variant of the Thompson construction that produces a 30-percent smaller machine. In Section 5.2, we briefly discuss five other constructions, each of which has some points of interest.

In all cases except one, we can prove that repeated null elimination on the resulting machines yields the corresponding Glushkov machines.

5.1 Sippu and Soisalon-Soininen's construction

In the first volume of their text on parsing, Sippu and Soisalon-Soininen [24] introduce a new inductive construction that produces a smaller machine than does the Thompson construction. More precisely, the S3 construction, as we call it, gives a machine that has size at most $2|E|$, for a regular expression E, whereas the Thompson construction gives a machine that has size at most $3|E|$. The construction is defined in Fig. 8. We provide an example of its use in Fig. 9 with the regular expression that is our running example.

Now, although S3 machines are homogeneous, they are not necessarily hammocks. The reason is the unusual induction step for the Kleene-star operation that introduces at least one vertex that is not on a simple source-sink path. Despite this failing, we can apply null elimination to an S3 machine to obtain a corresponding Glushkov machine. Since a formal proof of this fact is similar to that of Theorem 8, we illustrate, in Fig. 10, only the induction step for Kleene star. We summarize the claimed result in the following theorem.

Theorem 9 *For each empty-free regular expression E, if M_E^{S3} has exactly k distinct states $q_1, \ldots, q_k$ that satisfy the null-elimination precondition, then*

$$\mathrm{elim}(\cdots \mathrm{elim}(M_E^{S3}, q_1), \ldots, q_k) \cong M_E^G.$$

One final observation about the S3 construction is in order. The inductive definition of $F + G$ has an unpleasant side effect—it can produce states with unbounded in- and out-degrees. Thus, from a practical standpoint, what S3 machines gain in size over Thompson machines is more than lost by the increase in size of their states' in- and out-degrees. Thompson machines are, from a practical standpoint, pleasing to implement as not only are they small but also their states' in- and out-degrees are at most two.

5.2 Miscellaneous constructions

We discuss briefly five other inductive constructions that occur in textbooks. First, we have Lewis and Papadimitriou's [20] construction which is interesting in that the null-string symbol is not included in expressions; therefore, empty-free expressions do not denote either the empty language or the null-string language. As they use a single state that is a start state to represent the empty language, it is already Glushkov. Apart from this apparently minor variation their construction is Thompson-like as is that given by Wood [27]. As such null elimination yields Glushkov machines in both cases.

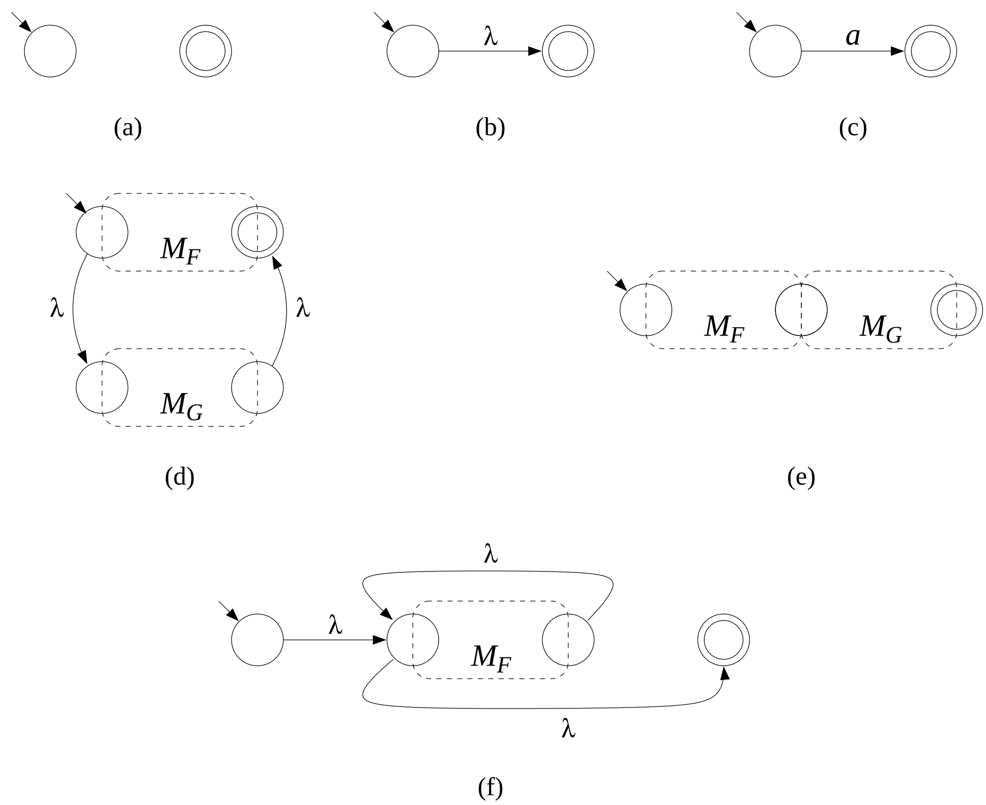

Figure 8: The S3 inductive construction. The finite-state machines correspond to the regular expressions: a. $E = \emptyset$; b. $E = \lambda$; c. $E = a$, $a \in \Sigma$; d. $E = (F + G)$; e. $E = (F \cdot G)$; and f. $E = (F^*)$.

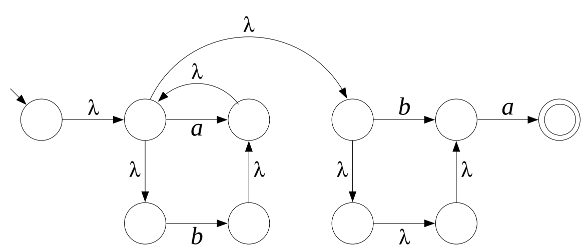

Figure 9: The application of the S3 construction to the running example expression.

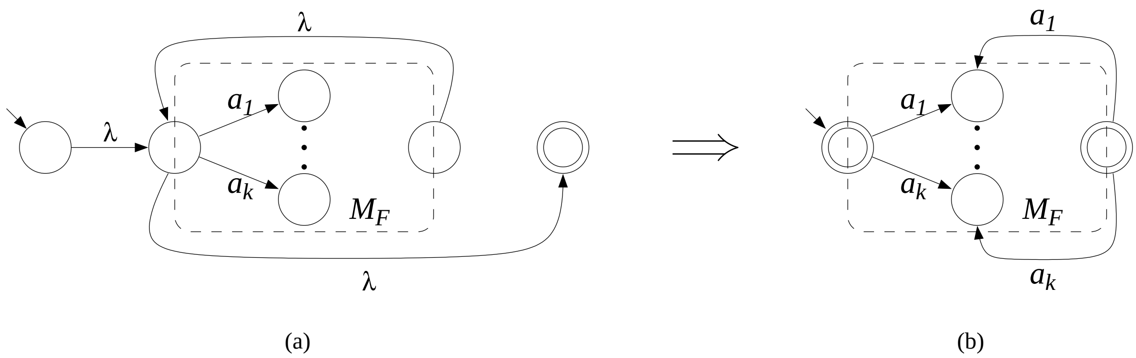

Figure 10: The S3 induction step for Kleene star.

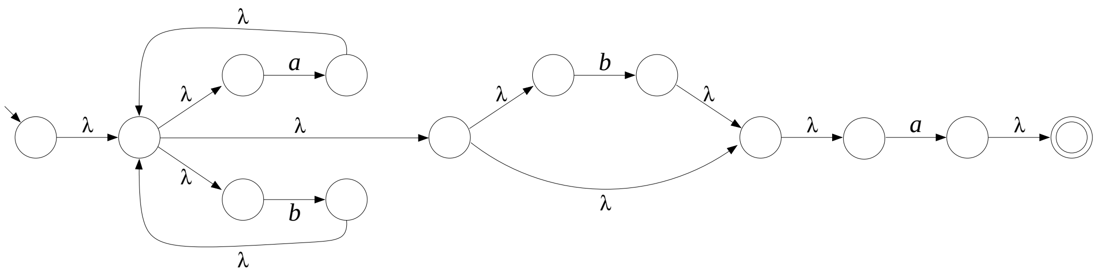

Figure 11: The result of applying the construction of Denning and his coauthors to the running example.

Second, we have the construction of Denning and his coauthors [11], which is also Thompson-like, but their induction step for Kleene star ensures that it does not always give a hammock; see Fig. 11. Their construction adds a null transition before and after a single-symbol transition in the base case and also for the induction step for the Kleene star. They also use the Thompson construction for the empty-set symbol and, hence, null elimination yields a Glushkov machine only when the given expression does not contain the empty-set symbol.

Third, Martin's [21] construction is also Thompson-like except that the Kleene-star step introduces a new start state that has null transitions both into and out of it; see Fig. 12. As a result it does not always produce a hammock.

Lastly, Drobot's [12] construction is interesting because it is almost the Champarnaud–Glushkov construction. It fails to meet that goal since the author provides an induction step for product that has four separate cases and two of them involve duplicating the second machine. As a result it is the only construction that we found that cannot be transformed into a Glushkov machine using null elimination.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.*

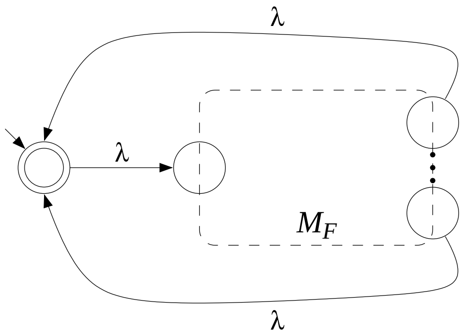

Figure 12: The induction step for the Kleene star operation in Martin's construction.

Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA, 1986.

[2] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972.

[3] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.

[4] A. Brüggemann-Klein and D. Wood. The validation of SGML content models. *Mathematical and Computer Modelling*, 25:73–84, 1997.

[5] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 140:229–253, 1998.

[6] P. Caron and D. Ziadi. Characterization of Glushkov automata. *Theoretical Computer Science*, 1998. To appear.

[7] J.-M. Champarnaud. AUTOMATE: Un système de manipulation des automates finis. Rapport LITP-85-48, Laboratoire d'Informatique Théorique de Paris, Université de Paris VII, 1985.

[8] J.-M. Champarnaud. From a regular expression to an automaton. Rapport LIR-97.08, Laboratoire d'Informatique de Rouen, Université de Rouen, France, 1997.

[9] J.-M. Champarnaud and G. Hansel. AUTOMATE: A computing package for automata and finite semigroups. *Journal of Symbolic Computation*, 12:197–220, 1991.

[10] J.-M. Champarnaud, J.-L. Ponty, and D. Ziadi. From regular expressions to finite automata. Unpublished manuscript, July 1998.

[11] P.J. Denning, J.B. Dennis, and J.E. Qualitz. *Machines, Languages, and Computation.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

[12] V. Drobot. *Formal Languages and Automata Theory.* Computer Science Press, San Francisco, CA, 1989.

[13] D. Giammarresi, J.-L Ponty, and D. Wood. A characterization of Thompson digraphs. Working manuscript, August 1998.

[14] V. M. Glushkov. On a synthesis algorithm for abstract automata. *Ukr. Matem. Zhurnal*, 12(2):147–156, 1960. In Russian.

[15] V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.

[16] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 2 edition, 1979.

[17] ISO 8879: Information processing—Text and office systems—Standard Generalized Markup Language (SGML), October 1986. International Organization for Standardization.

[18] V. Jansen, A. Potthoff, W. Thomas, and U. Wermuth. A short guide to the AMoRE system. Technical Report 90-02, Lehrstuhl für Informatik II, Universit—"at Aachen, Aachen, Germany, 1990.

[19] S.C. Kleene. Representation of events in nerve nets and finite automata. In C.E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42, Princeton, NJ, 1956. Princeton University Press.

[20] H.R. Lewis and C. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

[21] J.C. Martin. *Introduction to Languages and the Theory of Computation.* McGraw-Hill, New York, NY, 1991.

[22] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9:39–47, 1960.

[23] D.R. Raymond and D. Wood. Grail: A C++ library for automata and expressions. *Journal of Symbolic Computation*, 17:341–350, 1994.

[24] S. Sippu and E. Soisalon-Soininen. *Parsing Theory: I Languages and Parsing.* EATCS Monographs on Theoretical Computer Science, Volume 15. Springer-Verlag, New York, NY, 1988.

[25] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.

[26] B.W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1995.

[27] D. Wood. *Theory of Computation.* John Wiley & Sons, Inc., New York, NY, 1987.

[28] D. Wood. *Theory of Computation.* John Wiley & Sons, Inc., New York, NY, second edition, 1998. In preparation.