A taxonomy of finite automata construction algorithms*

Bruce W. Watson
Faculty of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
e-mail: `watson@win.tue.nl`
Tel: +31 40 474319

May 18, 1994

Abstract

This paper presents a taxonomy of finite automata construction algorithms. Each algorithm is classified into one of two families: those based upon the structure of regular expressions, and those based upon the automata-theoretic work of Myhill and Nerode.

Many of the algorithms appearing in the literature are based upon the structure of regular expressions. In this paper, we make this term precise by defining regular expressions as a Σ-term algebra, and automata constructions as various Σ-algebras of automata. Each construction algorithm is then presented as the unique natural homomorphism from the Σ-term algebra of regular expressions to the appropriate Σ-algebra of automata. The concept of duality is introduced and used to derive more practical construction algorithms. In this way, we successfully present (and relate) algorithms given by Thompson, Berry and Sethi, McNaughton and Yamada, Glushkov, and Aho, Sethi, and Ullman. Efficient implementations (including those due to Chang and Paige, and Brüggemann-Klein) are also treated. As a side-effect we derive several new algorithms.

A pair of impractical, but theoretically interesting, construction algorithms were presented by Myhill and Nerode. Some encoding techniques are used to make the algorithms practical — giving Brzozowski's algorithm based upon derivatives. DeRemer's algorithm is derived as an encoding of Brzozowski's algorithm. Two new algorithms, related to DeRemer's, are derived. Lastly, this family of algorithms is related to the first family.

In addition to classifying the algorithms, we identify (and abstract from) the coding tricks and implementation details present in many of the published algorithms. This paper also presents an introduction to finite automata, Σ-algebras, and their properties.

*Reprinted with corrections.

Contents

List of Figures

1 Introduction

The construction of finite automata (from regular expressions) is one of the oldest and most extensively developed areas of computing science. Just as the variety of applications has grown, so has the diversity of solutions. Some of the solutions were devised to deal with an extension of the problem, such as constructing a finite automaton from an extended regular expression[1], while others were devised with efficiency in mind. Such a myriad of objectives in the algorithm design has lead to solutions that are difficult to compare. Frequently, people that study the algorithms (or *constructions* as they are called in this paper) marvel that two seemingly different algorithms construct isomorphic finite automata from the same regular expression. In order to differentiate these algorithms, a taxonomy of construction algorithms would be useful. This report presents such a taxonomy. A related taxonomy of finite automata minimization algorithms appears in [Wats93].

In developing a taxonomy, we have the luxury of rearranging the relationships between the algorithms, possibly introducing relationships that are not present in the history of an algorithm's development. In this paper, for example, we derive DeRemer's construction from Myhill and Nerode's construction. Historically, the theory of LR parsing had a much greater influence on DeRemer's construction.

Section 2 gives definitions of finite automata and some transformations on them. Section 3 introduces Σ-algebras, the foundations for the first family of finite automata constructions. Sections 4 and 5 include the two families of finite automata constructions. Appendix A gives the basic definitions required for reading this paper, while Appendix B presents some proofs related to Section 4. The construction relationships are summarized in the "family tree" shown in Figure 1. The main results of the taxonomy are summarized in the conclusions — Section 6.

In this taxonomy, the finite automata constructions are arranged into two families: those constructions that are based upon the structure of regular expressions, and those based upon the automata-theoretic results of Myhill and Nerode.

The first family of constructions is presented in Section 4:

- Thompson's construction as presented in [Thom68]. This algorithm constructs a (possibly nondeterministic) finite automaton (possibly with ϵ-transitions). The description in this paper (Construction 4.3) is based upon those given in [AU92, HU79, Wood87, ASU86] as they are usually considered more readable than Thompson's original paper. Additionally, a more practical (top-down) version of Thompson's construction is presented (Construction 4.5).

- The ϵ-lookahead finite automaton construction. This algorithm (Construction 4.11) constructs finite automata that are similar to those constructed by Thompson's construction. They may include so-called ϵ-lookahead transitions.

- The guarded commands program construction. This algorithm (Construction 4.17) constructs a guarded commands program from a regular expression. The program is an acceptor for the regular language denoted by the regular expression. It is presented in this paper as a refinement (using hard-coded guarded commands) of the ϵ-lookahead construction.

- The left-biased and right-biased constructions. These two constructions (Constructions 4.22 and 4.43 respectively) are related by being the mirror images (or *duals*) of one another. They both construct an *ϵ-free* (possibly nondeterministic) finite automaton.

- Berry and Sethi's construction as presented in [BS86, Glus61, MY60]. This construction (Construction 4.32) uses some precomputation of sets to construct the same finite automaton as the left-biased construction. This construction is implicitly given by Glushkov [Glus61] and McNaughton and Yamada [MY60], where it is used as the nondeterministic finite automaton construction underlying a deterministic finite automaton construction. Berry and Sethi [BS86] explicitly present this algorithm, and they relate it to Brzozowski's construction

[1] An extended regular expression is one that includes either the intersection or complementation operator.

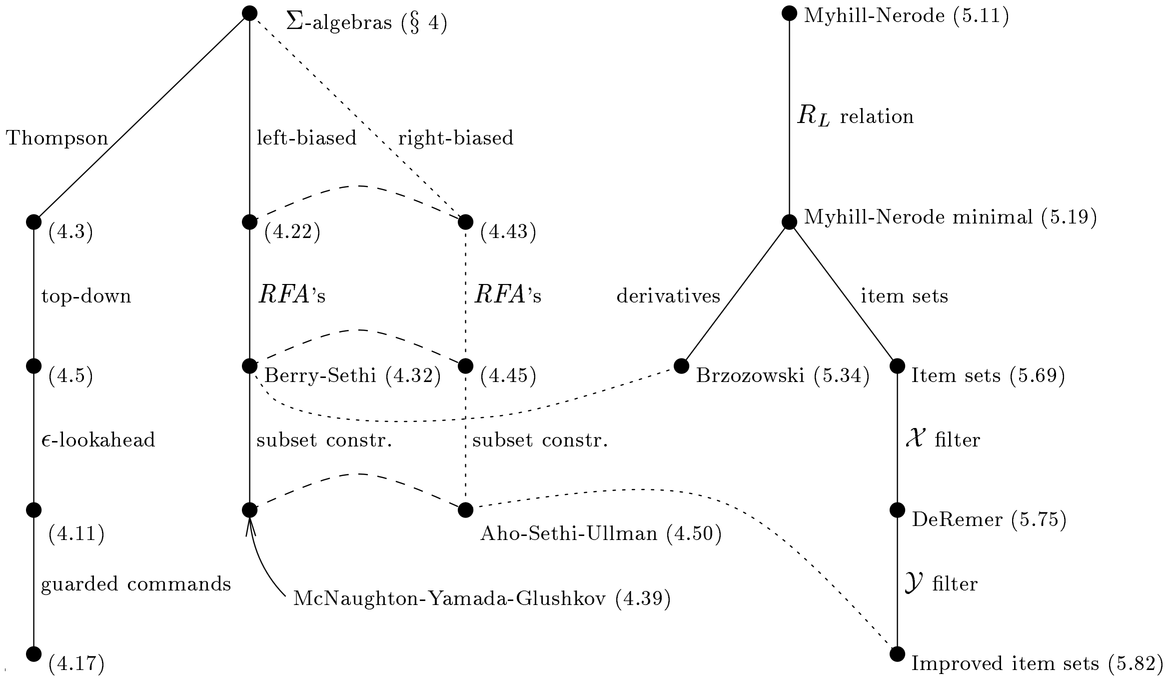


Figure 1: The family trees of finite automata constructions. The constructions fall into two broad categories: those based on the structure of regular expressions (descended from Σ-algebras), and those based on the Myhill-Nerode theorem. Each construction presented in this paper appears as a vertex in this tree, along with the name that it is given in this paper. If the construction is presented explicitly (in this paper), the construction number appears in parentheses (indicating where it appears in this paper). Solid edges denote refinements of the solution (and therefore explicit relationships between constructions). They are labeled with the name of the refinement. Dotted edges denote relationships (between algorithms) that are not elaborated upon in this paper. Some of the dotted edges are labeled with the name of the relationship or refinement. Vertices that are connected by a dashed edge are related by duality (they are the "mirror images" of one another).

[Brzo64]. We also present (in Construction 4.38) a variant of the Berry-Sethi construction that is more easily implemented in practice.

- McNaughton, Yamada and Glushkov's construction as presented in [MY60, Glus61]. This construction (Construction 4.39) produces a deterministic finite automaton.

- The dual of the Berry-Sethi construction. This construction (Construction 4.45) is the "mirror image" of Berry and Sethi's construction. A variant of this construction was also mentioned in passing by Aho, Sethi, and Ullman [ASU86, Example 3.22, pg. 140]; it appears in this paper as Construction 4.48. In our presentation of this construction, we correct an error appearing in Aho, Sethi, and Ullman's version (see Construction 4.48 of this paper).

- Aho, Sethi, and Ullman's construction as presented in [ASU86, Alg. 3.5, Fig. 3.44]. This construction (Construction 4.50) produces a deterministic finite automaton. It is the "mirror image" of the McNaughton-Yamada construction.

The second family of constructions (from regular expressions) are those based upon the automata-theoretic results of Myhill and Nerode [RS59]. They are presented in Section 5:

- Myhill and Nerode's construction as presented in [RS59]. This construction (Construction 5.11, which is given as part of the proof of the Myhill-Nerode theorem) uses some language theoretical results to construct a deterministic finite automaton. A version of this construction (Construction 5.19) gives the unique (up to isomorphism) minimal finite automaton. It is not a very practical construction (and usually is not even given as a construction), as it relies on the computation of possibly infinite sets. Certain encoding schemes can be used to represent these infinite sets, making the construction practical. Brzozowski's and DeRemer's constructions are two such encoding schemes.

- Brzozowski's construction as presented in [Brzo64]. This construction (Construction 5.34) gives a deterministic finite automaton. We derive it as an encoding of the Myhill-Nerode construction, although Brzozowski's derivation was entirely independent.

- The item set construction. This construction (Construction 5.69, not appearing in the literature) produces a deterministic finite automaton, and is based upon the concept of "items" which is borrowed from LR parsing [Knut65]. In this paper, we present it as an encoding of the Myhill and Nerode construction.

- DeRemer's construction as presented in [DeRe74]. This construction (Construction 5.75) produces a deterministic finite automaton. In this paper, it is derived from the item set construction, although DeRemer made use of LR parsing in his derivation.

- An improvement of the item set construction. This construction (Construction 5.82, not appearing in the literature) produces a deterministic finite automaton, and is also based upon the item set construction. Furthermore, it is an improvement of DeRemer's construction. A variant (Construction 5.85) is also related to the Aho-Sethi-Ullman deterministic finite automaton construction.

2 Finite automata

In this section we define finite automata, some of their properties, and some transformations on finite automata.

Definition 2.1 (Finite automaton): A finite automaton (an *FA*) is a 6-tuple (Q, V, T, E, S, F) where

- Q is a finite set of states,
- V is an alphabet,
- $T \in \mathcal{P}(Q \times V \times Q)$ is a transition relation,
- $E \in \mathcal{P}(Q \times Q)$ is an ϵ-transition relation
- $S \subseteq Q$ is a set of start states, and
- $F \subseteq Q$ is a set of final states.

The definitions of an alphabet and function $\mathcal{P}$ are in Definition A.9 and Convention A.1 respectively. □

Remark 2.2: We will take some liberty in our interpretation of the signatures of the transition relations. For example, we also use the signatures $T \in V \longrightarrow \mathcal{P}(Q \times Q)$, $T \in Q \times Q \longrightarrow \mathcal{P}(V)$, $T \in Q \times V \longrightarrow \mathcal{P}(Q)$, $T \in Q \longrightarrow \mathcal{P}(V \times Q)$, and $E \in Q \longrightarrow \mathcal{P}(Q)$. In each case, the order of the Q's from left to right will be preserved; for example, the function $T \in Q \longrightarrow \mathcal{P}(V \times Q)$ is defined as $T(p) = \{(a, q) : (p, a, q) \in T\}$. The signature that is used will be clear from the context. See Remark A.3. The definition of $\longrightarrow$ appears in Convention A.2. □

Remark 2.3: Our definition of finite automata differs from the traditional approach in three ways:

- multiple start states are permitted;
- the transition relations are presented in a symmetrical way (without any inherent left-to-right bias); and
- the ϵ-transitions (relation E) are separate from transitions on alphabet symbols (relation T).

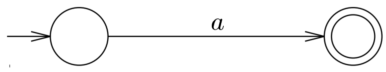
□

Since we only consider finite automata in this paper, we will frequently simply use the term *automata*.

Convention 2.4 (Finite automaton state graphs): When drawing the state graph corresponding to a finite automaton, we adopt the following conventions:

- All states are drawn as circles (vertices).
- Transitions are drawn as labeled (with ϵ or alphabet symbol $a \in V$) directed edges between states.
- Start states have an in-transition with no source (the transition does not come from another state).
- Final states are drawn as two concentric circles.

For example, the *FA* below has two states, one is the start state, and other is the final state, with a transition on a:



□

2.1 Properties of finite automata

In this subsection we define some properties of finite automata. To make these definitions more concise, we introduce particular finite automata $M = (Q, V, T, E, S, F)$, $M_0 = (Q_0, V_0, T_0, E_0, S_0, F_0)$, and $M_1 = (Q_1, V_1, T_1, E_1, S_1, F_1)$.

Definition 2.5 (Size of an *FA*): Define the size of an *FA* as $|M| = |Q|$. $\Box$

Definition 2.6 (Isomorphism ($\cong$) of *FA*'s): We define isomorphism ($\cong$) as an equivalence relation on *FA*'s. M_0 and M_1 are isomorphic (written $M_0 \cong M_1$) if and only if $V_0 = V_1$ and there exists a bijection $g \in Q_0 \longrightarrow Q_1$ such that

- $T_1 = \{(g(p), a, g(q)) : (p, a, q) \in T_0\}$,
- $E_1 = \{(g(p), g(q)) : (p, q) \in E_0\}$,
- $S_1 = \{g(s) : s \in S_0\}$, and
- $F_1 = \{g(f) : f \in F_0\}$.

$\Box$

Definition 2.7 (Extending the transition relation T): We extend transition relation $T \in V \longrightarrow \mathcal{P}(Q \times Q)$ to $T^* \in V^* \longrightarrow \mathcal{P}(Q \times Q)$ as follows:

$$T^*(\epsilon) = E^*$$

and (for $a \in V, w \in V^*$)

$$T^*(aw) = E^* \circ T(a) \circ T^*(w)$$

Operator $\circ$ (composition) is defined in Convention A.6. This definition could also have been presented symmetrically. $\Box$

Remark 2.8: We also sometimes use the signature $T^* \in Q \times Q \longrightarrow \mathcal{P}(V^*)$. $\Box$

Remark 2.9: If $E = \emptyset$ then $E^* = \emptyset^* = I_Q$ where I_Q is the identity relation on the states of M. $\Box$

Definition 2.10 (The language between states): The language between any two states $q_0, q_1 \in Q$ is $T^*(q_0, q_1)$. $\Box$

Definition 2.11 (Left and right languages): The left language of a state (in M) is given by function $\overleftarrow{\mathcal{L}}_M \in Q \longrightarrow \mathcal{P}(V^*)$, where

$$\overleftarrow{\mathcal{L}}_M(q) = (\cup\ s : s \in S : T^*(s, q))$$

The right language of a state (in M) is given by function $\overrightarrow{\mathcal{L}}_M \in Q \longrightarrow \mathcal{P}(V^*)$, where

$$\overrightarrow{\mathcal{L}}_M(q) = (\cup\ f : f \in F : T^*(q, f))$$

The subscript M is usually dropped when no ambiguity can arise. $\Box$

Definition 2.12 (Language of an *FA*): The language of a finite automaton (with alphabet V) is given by the function $\mathcal{L}_{FA} \in FA \longrightarrow \mathcal{P}(V^*)$ defined as:

$$\mathcal{L}_{FA}(M) = (\cup\ s, f : s \in S \wedge f \in F : T^*(s, f))$$

$\Box$

Property 2.13 (Language of an *FA*): From the definitions of left and right languages (of a state), we can also write:

$$\mathcal{L}_{FA}(M) = (\cup\ f : f \in F : \overleftarrow{\mathcal{L}}(f))$$

and

$$\mathcal{L}_{FA}(M) = (\cup\ s : s \in S : \overrightarrow{\mathcal{L}}(s))$$

□

Definition 2.14 (Extension of $\mathcal{L}_{FA}$): Function $\mathcal{L}_{FA}$ is extended to $[FA]_{\cong}$ as $\mathcal{L}_{FA}([M]_{\cong}) = \mathcal{L}_{FA}(M)$. This use of brackets ([,]) is defined in Convention A.7. The choice of representative is irrelevant, as isomorphic *FA*'s accept the same language. □

Definition 2.15 (*Complete*): A *Complete* finite automaton is one satisfying the following:

$$Complete(M) \equiv (\forall\ q, a : q \in Q \wedge a \in V : T(q, a) \neq \emptyset)$$

□

Property 2.16 (*Complete*): For all *Complete* *FA*'s (Q, V, T, E, S, F):

$$(\cup\ q : q \in Q : \overleftarrow{\mathcal{L}}(q)) = V^*$$

□

Definition 2.17 (ϵ-*free*): Automaton M is ϵ-*free* if and only if $E = \emptyset$. □

Remark 2.18: Even if M is ϵ-*free* it is still possible that $\epsilon \in \mathcal{L}_{FA}(M)$: in this case $S \cap F \neq \emptyset$. □

Definition 2.19 (Reachable states): For M we can define a reachability relation $Reach(M) \subseteq (Q \times Q)$ defined as

$$Reach(M) = (\bar{\pi}_2(T) \cup E)^*$$

Functions π and $\bar{\pi}$ are defined in Convention A.4. Similarly the set of start-reachable states is defined to be:

$$SReachable(M) = Reach(M)(S)$$

and the set of final-reachable states is defined to be:

$$FReachable(M) = (Reach(M))^R(F)$$

Reversal of a relation is defined in Definition A.20. The set of useful states is:

$$Reachable(M) = SReachable(M) \cap FReachable(M)$$

□

Remark 2.20: For *FA* $M = (Q, V, T, E, S, F)$, function *SReachable* satisfies the following interesting property:

$$q \in SReachable(M) \equiv \overleftarrow{\mathcal{L}}_M(q) \neq \emptyset$$

FReachable satisfies a similar property:

$$q \in FReachable(M) \equiv \overrightarrow{\mathcal{L}}_M(q) \neq \emptyset$$

□

Definition 2.21 (Useful automaton): A *Useful* finite automaton is one with only reachable states:

$$Useful(M) \equiv (Q = Reachable(M))$$

□

Definition 2.22 (Start-useful automaton): A $Useful_s$ finite automaton is one with only start-reachable states:

$$Useful_s(M) \equiv (Q = SReachable(M))$$

□

Definition 2.23 (Final-useful automaton): A $Useful_f$ finite automaton is one with only final-reachable states

$$Useful_f(M) \equiv (Q = FReachable(M))$$

□

Remark 2.24: $Useful_s$ and $Useful_f$ are closely related by *FA* reversal (to be presented in Transformation 2.34). For all $M \in FA$ we have $Useful_f(M) \equiv Useful_s(M^R)$. □

Property 2.25 (Deterministic finite automaton): A finite automaton M is deterministic if and only if

- it does not have multiple start states,
- it is ϵ-*free*, and
- transition function $T \in Q \times V \longrightarrow \mathcal{P}(Q)$ does not map pairs in $Q \times V$ to multiple states.

Formally,

$$Det(M) \equiv (|S| \leq 1 \wedge \epsilon\text{-}free(E) \wedge (\forall\ q, a : q \in Q \wedge a \in V : |T(q,a)| \leq 1))$$

□

Definition 2.26 (Deterministic *FA*'s): *DFA* denotes the set of all deterministic finite automata. We call $FA \setminus DFA$ the set of *nondeterministic finite automata.* □

Convention 2.27 (Transition function of a *DFA*): For $(Q, V, T, \emptyset, S, F) \in DFA$ we can consider the transition function to have signature $T \in Q \times V \not\longrightarrow Q$. (A definition of $\not\longrightarrow$ appears in Convention A.2.) The transition function is total if and only if the *DFA* is *Complete*. □

Property 2.28 (Weakly deterministic automaton): Some authors use a definition of a deterministic automaton that is weaker than *Det*; it uses left languages and is defined as follows:

$$Det'(M) \equiv (\forall\ q_0, q_1 : q_0 \in Q \wedge q_1 \in Q \wedge q_0 \neq q_1 : \overleftarrow{\mathcal{L}}(q_0) \cap \overleftarrow{\mathcal{L}}(q_1) = \emptyset)$$

□

Remark 2.29: $Det(M) \Rightarrow Det'(M)$ is easily proved. We can also demonstrate that there exists an $M \in FA$ such that $Det'(M) \wedge \neg Det(M)$:

$$(\{q_0, q_1\}, \{b\}, \{(q_0, b, q_0), (q_0, b, q_1)\}, \emptyset, \emptyset, \emptyset)$$

In this *FA*, $\overleftarrow{\mathcal{L}}(q_0) = \overleftarrow{\mathcal{L}}(q_1) = \emptyset$, but state q_0 has two out-transitions on symbol alphabet b. □

Definition 2.30 (Minimality of a ***DFA*****):** An $M \in DFA$ is minimal as follows:

$$Min(M) \equiv (\forall\ M' : M' \in DFA \wedge \mathcal{L}_{FA}(M) = \mathcal{L}_{FA}(M') : |M| \leq |M'|)$$

Predicate *Min* is defined only on *DFA*'s. Some definitions are simpler if we define a minimal, but still *Complete*, *DFA* as follows:

$$Min_C(M) \equiv (\forall\ M' : M' \in DFA \wedge Complete(M') \wedge \mathcal{L}_{FA}(M) = \mathcal{L}_{FA}(M') : |M| \leq |M'|)$$

Predicate Min_C is defined only on *Complete* *DFA*'s. □

Property 2.31 (Minimality of a ***DFA*****):** An M, such that $Min(M)$, is the unique (modulo $\cong$) minimal *DFA*, as will be shown in Section 5. There is no similar uniqueness property for nondeterministic finite automata. □

Property 2.32 (An alternate definition of minimality of a ***DFA*****):** For the purposes of minimizing a *DFA*, we use the definition (defined only on *DFA*'s):

$$\begin{aligned} Minimal(Q,V,T,\emptyset,S,F) \quad \equiv \quad & (\forall\ q_0,q_1 : q_0 \in Q \wedge q_1 \in Q \wedge q_0 \neq q_1 : \overrightarrow{\mathcal{L}}(q_0) \neq \overrightarrow{\mathcal{L}}(q_1)) \\ & \wedge\ Useful(Q,V,T,\emptyset,S,F) \end{aligned}$$

We have the property that (for all $M \in DFA$) $Minimal(M) \equiv Min(M)$. It is easy to prove that $Min(M) \Rightarrow Minimal(M)$. The reverse direction follows from the Myhill-Nerode theorem (Theorem 5.7).

A similar definition that relates to Min_C is (also defined only on *DFA*'s):

$$\begin{aligned} Minimal_C(Q,V,T,\emptyset,S,F) \quad \equiv \quad & (\forall\ q_0,q_1 : q_0 \in Q \wedge q_1 \in Q \wedge q_0 \neq q_1 : \overrightarrow{\mathcal{L}}(q_0) \neq \overrightarrow{\mathcal{L}}(q_1)) \\ & \wedge\ Useful_s(Q,V,T,\emptyset,S,F) \end{aligned}$$

We have the property that (for all $M \in DFA$ such that $Complete(M)$) $Minimal_C(M) \equiv Min_C(M)$. The contrapositive of $Min_C(M) \Rightarrow Minimal_C(M)$ is easily proved, and the reverse direction also follows from Theorem 5.7. □

Remark 2.33: In the literature the second conjunct in the definition of predicate $Minimal_C$ is sometimes erroneously omitted. The necessity of the conjunct can be seen by considering the *DFA*

$$(\{p,q\},\{a\},\{(p,a,p),(q,a,q)\},\emptyset,\emptyset,\{p\})$$

Here $\overleftarrow{\mathcal{L}}(p) = \overleftarrow{\mathcal{L}}(q) = \emptyset$ (which is also the language of the *DFA*), $\overrightarrow{\mathcal{L}}(p) = \{a\}^*$, and $\overrightarrow{\mathcal{L}}(q) = \emptyset$. Without the second conjunct, this *DFA* would be considered $Minimal_C$; clearly this is not the case, as the minimal *Complete* *DFA* accepting $\emptyset$ is $(\emptyset,\{a\},\emptyset,\emptyset,\emptyset,\emptyset)$. □

2.2 Transformations on finite automata

Transformation 2.34 (***FA*** **reversal):** *FA* reversal is given by postfix (superscript) function $R \in FA \longrightarrow FA$, defined as:

$$(Q,V,T,E,S,F)^R = (Q,V,T^R,E^R,F,S)$$

Function R satisfies

$$(\forall\ M : M \in FA : (\mathcal{L}_{FA}(M))^R = \mathcal{L}_{FA}(M^R)).$$

and preserves ϵ-*free* and *Useful*.

Reversal functions are defined in Definition A.19, and preservation is defined in Definition A.18. □

Remark 2.35: The property $(\mathcal{L}_{FA}(M^R))^R = \mathcal{L}_{FA}(M)$ means that function $\mathcal{L}_{FA}$ is its own dual, and is therefore symmetrical (see Definitions A.21 and A.22). □

Definition 2.36 (Extending reversal to $[FA]_{\cong}$): We extend reversal to $R \in [FA]_{\cong} \longrightarrow [FA]_{\cong}$ defined as $([M]_{\cong})^R = [M^R]_{\cong}$. The definition is independent of the choice of representative (of an equivalence class of $\cong$) since R and isomorphism commute. □

Transformation 2.37 (Useless state removal): There exists a function $useful \in FA \longrightarrow FA$ that removes states that are not reachable. A definition of this function is not given here, as it is not needed. Function *useful* satisfies

$$(\forall\ M : M \in FA : Useful(useful(M)) \wedge \mathcal{L}_{FA}(useful(M)) = \mathcal{L}_{FA}(M))$$

and can be defined so as to preserve *ϵ-free*, *Useful*, *Det*, and *Min*. □

Transformation 2.38 (Removing start state unreachable states): Transformation $useful_s \in FA \longrightarrow FA$ removes those states that are not start-reachable:

$$\begin{array}{lcll} useful_s(Q,V,T,E,S,F) & = & \textbf{let} & U = SReachable(Q,V,T,E,S,F) \\ & & \textbf{in} & \\ & & & (U, V, T \cap (U \times V \times U), E \cap (U \times U), S \cap U, F \cap U) \\ & & \textbf{end} & \end{array}$$

Function $useful_s$ satisfies

$$(\forall\ M : M \in FA : Useful_s(useful_s(M)) \wedge \mathcal{L}_{FA}(useful_s(M)) = \mathcal{L}_{FA}(M))$$

and preserves *Complete*, *ϵ-free*, *Useful*, *Det*, and (trivially) Min_C and *Min*. □

Remark 2.39: A function $useful_f \in FA \longrightarrow FA$ could also be defined, removing states that are not final-reachable. Such a function is not needed in this paper. □

Transformation 2.40 (Completing an *FA*): Function $complete \in FA \longrightarrow FA$ is defined as:

$$\begin{array}{lcll} complete(Q,V,T,E,S,F) & = & \textbf{let} & s \text{ be a new (sink) state} \\ & & \textbf{in} & \\ & & & \begin{array}{ll} \textbf{let} & T' = \{(p,a,s) : (\nexists\ q : q \in Q : (p,a,q) \in T)\} \\ & T'' = \textbf{if } (T' \neq \emptyset) \textbf{ then } \{s\} \times V \times \{s\} \textbf{ else } \emptyset \textbf{ fi} \\ \textbf{in} & \\ & (Q \cup \textbf{if } (T' \neq \emptyset) \textbf{ then } \{s\} \textbf{ else } \emptyset \textbf{ fi}, V, \\ & \quad T \cup T' \cup T'', E, S, F) \\ \textbf{end} & \end{array} \\ & & \textbf{end} & \end{array}$$

It satisfies the requirement that:

$$(\forall\ M : M \in FA : Complete(complete(M)) \wedge \mathcal{L}_{FA}(complete(M)) = \mathcal{L}_{FA}(M))$$

In general, this transformation adds a sink state to the *FA*. This transformation preserves *ϵ-free*, (trivially) *Complete*, *Det*, and Min_C. □

Transformation 2.41 (ϵ removal): An ϵ removal transformation $remove_\epsilon \in FA \longrightarrow FA$ is one that satisfies

$$(\forall\ M : M \in FA : \epsilon\text{-}free(remove_\epsilon(M)) \wedge \mathcal{L}_{FA}(remove_\epsilon(M)) = \mathcal{L}_{FA}(M))$$

There are several possible implementations of $remove_\epsilon$. One implementation is:

$$\begin{array}{lcll} remove_{\epsilon,sym}(Q,V,T,E,S,F) & = & \textbf{let} & T'(a) = E^* \circ T(a) \circ E^* \\ & & \textbf{in} & \\ & & & (Q, V, T', \emptyset, E^*(S), (E^*)^R(F)) \\ & & \textbf{end} & \end{array}$$

This implementation preserves *Complete* and *Useful* and is symmetrical. □

Transformation 2.42 (Subset construction): The function *subset* transforms an ϵ*-free FA* into a *DFA* (in the **let** clause $T' \in \mathcal{P}(Q) \times V \longrightarrow \mathcal{P}(\mathcal{P}(Q))$)

$$\begin{array}{lcll} subset(Q, V, T, \emptyset, S, F) & = & \textbf{let} & T'(U, a) = \{(\cup\ q : q \in U : T(q, a))\} \\ & & & F' = \{U : U \in \mathcal{P}(Q) \wedge U \cap F \neq \emptyset\} \\ & & \textbf{in} & \\ & & & (\mathcal{P}(Q), V, T', \emptyset, \{S\}, F') \\ & & \textbf{end} & \end{array}$$

In addition to the obvious property that (for all $M \in FA$) $\mathcal{L}_{FA}(subset(M)) = \mathcal{L}_{FA}(M)$, function *subset* satisfies

$$(\forall\ M : M \in FA \wedge \epsilon\text{-}free(M) : Det(subset(M)) \wedge Complete(subset(M)))$$

and preserves *Complete*, ϵ*-free*, *Det*, and Min_C. It is also known as the "powerset" construction. □

Property 2.43 (Subset construction): Let $M_0 = (Q_0, V, T_0, \emptyset, S_0, F_0)$ and $M_1 = subset(M_0)$ be finite automata. By the subset construction, the state set of M_1 is $\mathcal{P}(Q_0)$. We have the following property:

$$(\forall\ p : p \in \mathcal{P}(Q_0) : \overrightarrow{\mathcal{L}}_{M_1}(p) = (\cup\ q : q \in p : \overrightarrow{\mathcal{L}}_{M_0}(q)))$$

□

Definition 2.44 (Optimized subset construction): The function *subsetopt* transforms an ϵ*-free FA* into a *DFA*. This function is an optimized version of *subset*.

$$\begin{array}{lcll} subsetopt(Q, V, T, \emptyset, S, F) & = & \textbf{let} & T'(U, a) = \{(\cup\ q : q \in U : T(q, a))\} \\ & & & Q' = \mathcal{P}(Q) \setminus \{\emptyset\} \\ & & & F' = \{U : U \in \mathcal{P}(Q) \wedge U \cap F \neq \emptyset\} \\ & & \textbf{in} & \\ & & & (Q', V, T' \cap (Q' \times V \times Q'), \emptyset, \{S\}, F') \\ & & \textbf{end} & \end{array}$$

In addition to the property that (for all $M \in FA$) $\mathcal{L}_{FA}(subsetopt(M)) = \mathcal{L}_{FA}(M)$, function *subsetopt* satisfies

$$(\forall\ M : M \in FA \wedge \epsilon\text{-}free(M) : Det(subsetopt(M)))$$

and preserves ϵ*-free*, *Det*. □

2.2.1 Algorithms implementing the subset construction

Since many of the states in a *subset*-constructed *DFA* may be unreachable, we consider an algorithm implementing the composition $useful_s \circ subset$.

In this algorithm, D (for done) is the set of states (of the *DFA* being constructed) already considered, and U (for un-done) is the set of states to be considered. The type of S', D, and U is $\mathcal{P}(\mathcal{P}(Q))$ (in particular, S' is a set of states in the constructed *DFA*). This algorithm will yield a *Complete DFA*. In the case that the language of the automaton (being subset constructed) is not V^*, then there will be a state $\emptyset \in D$ which is the sink state. The algorithm is implemented in Dijkstra's guarded command language [Dijk76].

Algorithm 2.45:

$\{(Q, V, T, \emptyset, S, F) \in FA\}$
$S', T' := \{S\}, \emptyset;$
$D, U := \emptyset, S';$
do $U \neq \emptyset \longrightarrow$
  let $u : u \in U;$
  $D, U := D \cup \{u\}, U \setminus \{u\};$
  for $a : a \in V$ **do**
    $d := (\cup\ q : q \in u : T(q, a));$
    if $d \notin D \longrightarrow U := U \cup \{d\}$
    $[\!]\ d \in D \longrightarrow$ **skip**
    fi;
    $T' := T' \cup \{(u, a, d)\}$
  rof
od;
$F' := \{d : d \in D \wedge d \cap F \neq \emptyset\}$
$\{(D, V, T', \emptyset, S', F') = \mathit{useful}_s \circ \mathit{subset}(Q, V, T, \emptyset, S, F)\}$
$\{\mathit{Complete}(D, V, T', \emptyset, S', F')\}$

An algorithm implementing $\mathit{useful}_s \circ \mathit{subsetopt}$, yielding a (possibly non-*Complete*) *DFA* with no sink state is:

Algorithm 2.46:

$\{(Q, V, T, \emptyset, S, F) \in FA\}$
$S', T' :=$ (**if** $(S \neq \emptyset)$ **then** $\{S\}$ **else** $\emptyset$ **fi**)$, \emptyset;$
$D, U := \emptyset, S';$
do $U \neq \emptyset \longrightarrow$
  let $u : u \in U;$
  $D, U := D \cup \{u\}, U \setminus \{u\};$
  for $a : a \in V \wedge (\exists\ q : q \in u : T(q, a) \neq \emptyset)$ **do**
    $d := (\cup\ q : q \in u : T(q, a));$
    if $d \notin D \longrightarrow U := U \cup \{d\}$
    $[\!]\ d \in D \longrightarrow$ **skip**
    fi;
    $T' := T' \cup \{(u, a, d)\}$
  rof
od;
$F' := \{d : d \in D \wedge d \cap F \neq \emptyset\}$
$\{(D, V, T', \emptyset, S', F') = \mathit{useful}_s \circ \mathit{subsetopt}(Q, V, T, \emptyset, S, F)\}$

Remark 2.47: The algorithm given above can be made more efficient by removing the $\exists$ quantification from the **for** guard, and implementing it in an **if-fi** structure within the **for** statement. The algorithm is left in this form since it is used in Construction 4.50 to present the Aho-Sethi-Ullman algorithm. $\square$

3 Σ-algebras and regular expressions

Many of the known *FA* constructions have definitions that follow the syntactic structure of regular expressions. The best known (and perhaps the easiest to understand) is Thompson's construction [Thom68]. We would like to formalize the notion of "following the syntactic structure." This is done by introducing Σ-algebras in this section. Regular expressions are then defined as a Σ-algebra.

3.1 Some basic definitions

This subsection provides the basic definitions required for Σ-algebras. Most of these definitions are taken, with slight modification, from [EM85].

Definition 3.1 (Sorts): Given set S (the elements of which are called *sorts*), a set of sets X is called S-sorted if the elements of X correspond one-to-one with S. The element of X corresponding to $s \in S$ is written X_s. □

Definition 3.2 (Signature Σ): A *signature* Σ is a pair (S, Γ) where

- S is a finite set, and
- Γ is an $(S^* \times S)$-sorted set called the *operators*.

We write elements of $S^* \times S$ as $<< s_1, \ldots, s_k >, s >$. We can make a couple of notational simplifications:

- Given $\gamma \in \Gamma_{<<s_1,\ldots,s_k>,s>}$ we write $\gamma : s_1 \times \ldots \times s_k \longrightarrow s$. Constant k is known as the *arity* of operator γ.
- For $\gamma \in \Gamma_{<<>,s>}$ we write $\gamma : s$, and call γ a *constant*; that is, constants are operators of arity zero.

□

Remark 3.3: Although the set $S^* \times S$ is infinite (for $S \neq \emptyset$), this does not imply that there are infinitely many operators. There may be $<< s_1, \ldots, s_m >, s > \in S^* \times S$ such that $\Gamma_{<<s_1,\ldots,s_m>,s>} = \emptyset$; in that case, there is no operator $\gamma : s_1 \times \ldots \times s_m \longrightarrow s$. □

Several of the following definitions are with respect to signature $\Sigma = (S, \Gamma)$.

Definition 3.4 ($Term_\Sigma$): The S-sorted set $Term_\Sigma$ is the smallest S-sorted set such that[2]

- if $\gamma : s_1 \times \ldots \times s_k \longrightarrow s$ (for some $k \geq 0$) $(s, s_1, \ldots, s_k \in S)$ and (for all $1 \leq i \leq k$) $t_i \in Term_{\Sigma_{s_i}}$ then $\gamma[t_1, \ldots, t_k] \in Term_{\Sigma_s}$. We adopt the convention that $\gamma[]$ is simply written γ.

□

Definition 3.5 (Σ-algebra): A Σ-algebra is a pair (V, F) such that

- V is an S-sorted set, and
- F is a set of functions f_γ (with $\gamma \in \Gamma$) such that
  - if $\gamma : s_1 \times \ldots \times s_k \longrightarrow s$ then $f_\gamma \in V_{s_1} \times \ldots \times V_{s_k} \longrightarrow V_s$.

Set V is called the *carrier set* of the Σ-algebra. Set F is called the *operator set* of the Σ-algebra. □

[2]Square brackets ([and]) are used syntactically here.

Definition 3.6 (Σ-term algebra): The Σ-term algebra is the Σ-algebra $(Term_\Sigma, F)$ such that

$$F = \{f_\gamma : (\gamma : s_1 \times \ldots \times s_k \longrightarrow s)\}$$

where (for all $f_\gamma \in F$) $f_\gamma \in Term_{\Sigma_{s_1}} \times \ldots \times Term_{\Sigma_{s_k}} \longrightarrow Term_{\Sigma_s}$ is defined as $f_\gamma(t_1, \ldots, t_k) = \gamma[t_1, \ldots, t_k]$. $\square$

Definition 3.7 (Σ-homomorphism): Given Σ-algebras (V, F) and (W, G), a Σ-homomorphism from (V, F) to (W, G) is an S-indexed set of functions h such that

- for all $s \in S$ we have $h_s \in V_s \longrightarrow W_s$, and
- for all $\gamma : s_1 \times \ldots \times s_k \longrightarrow s$, $f_\gamma \in F$, $g_\gamma \in G$, and $e_1 \in V_{s_1}, \ldots, e_k \in V_{s_k}$

$$h_s(f_\gamma(e_1, \ldots, e_k)) = g_\gamma(h_{s_1}(e_1), \ldots, h_{s_k}(e_k))$$

$\square$

Remark 3.8: In the case that there is only one sort, a Σ-homomorphism is a singleton set and we speak of the homomorphic function. $\square$

Definition 3.9 (Initial Σ-algebra): A Σ-algebra is *initial* if there is a unique Σ-homomorphism from it to all other Σ-algebras. $\square$

Proposition 3.10 (Σ-term algebras): Σ-term algebras are initial. $\square$

Example 3.11 (Σ-algebras): Consider signature $\Sigma = (S, \Gamma)$ where S consists only of sort $expr$, and Γ consists of constant $a : expr$ and operator $plus : expr \times expr \longrightarrow expr$. Some examples of terms in the Σ-term algebra are $plus[a, a]$ and $plus[plus[a, plus[a, a]], a]$.

We define another Σ-algebra X with the natural numbers as the carrier set, 0 (the natural number) as the constant, and $f_{plus}(x, y) = (x \,\mathbf{max}\, y) + 1$ as the operator.

As an example of a Σ-homomorphism, we define the "expression tree height" function as a homomorphism from the Σ-term algebra to algebra X. With only one sort, we define function h_{expr} as $h_{expr}(a) = 0$ and $h_{expr}(plus[e, f]) = f_{plus}(h_{expr}(e), h_{expr}(f)) = (h_{expr}(e) \,\mathbf{max}\, h_{expr}(f)) + 1$. $\square$

3.2 Regular expressions as a Σ-term algebra

Definition 3.12 (Regular expressions): We define *regular expressions* (over alphabet V) as the Σ-term algebra over signature $\Sigma = (S, O)$ where

- S consists of a single sort *Reg* (for regular expression), and
- O is a set of several constants: $\epsilon, \emptyset, a_1, \ldots, a_n$: *Reg* (where $V = \{a_1, \ldots, a_n\}$) and five operators $\cdot : Reg \times Reg \longrightarrow Reg$ (the dot operator), $\cup : Reg \times Reg \longrightarrow Reg$, $* : Reg \longrightarrow Reg$, $+ : Reg \longrightarrow Reg$, and $? : Reg \longrightarrow Reg$.

Signature Σ will be used throughout the remainder of this paper. We make the following notational simplification when writing terms in the Σ-term algebra:

- operators $\cdot$ (the dot) and $\cup$ are written as infix operators;
- operator $\cdot$ is usually not written, juxtaposition is used instead;
- operators $*$, $+$, and ? are written as postfix (superscript) operator.

The following will also be used for conciseness:

- a term in the Σ-term algebra is called a regular expression;

- the set $Term_\Sigma$ is denoted by *RE*;
- the operators have (ascending) precedence: $\cup$, $\cdot$, $*$ and $+$ and $?$; ϵ, $\emptyset$, and $a_1, \ldots, a_n \in V$ are constants;
- regular expressions are usually fully parenthesized; parentheses can be omitted where the operator precedence allows.

□

Remark 3.13: The ? operator is non-standard. It will be used to denote union with the language containing the empty string ϵ. See Definition 3.17. □

Remark 3.14: Some authors write $\cup$ as (infix) $+$ or as $|$. □

Example 3.15 (A regular expression): Given alphabet $V = \{a, b\}$ the regular expression $\cdot[\cup[a, \epsilon], *[b]]$ is usually written as $(a \cup \epsilon)b^*$. This particular regular expression will be used in running examples of *FA* construction. □

Remark 3.16: Some authors leave $\emptyset$, ?, or $+$ out of the definition of regular expressions. Strictly speaking, operators ϵ, $+$, and ? are not needed in the signature, since they can be constructed from the other operators. There are some *FA* constructions (from *RE*'s) that have running time dependent on the size of the regular expression. In these cases, treating the extra operators fully (instead of as abbreviations) becomes advantageous. □

Definition 3.17 (The Σ-algebra of regular languages): We define a Σ-algebra of regular languages (over alphabet V), with carrier $\mathcal{P}(V^*)$ and constants:

- $\{\epsilon\} \in \mathcal{P}(V^*)$ (the language containing only the empty string);
- $\emptyset \in \mathcal{P}(V^*)$ (the empty language);
- $\{a\} \in \mathcal{P}(V^*)$ (for all $a \in V$).

and operators:

- $\cup \in \mathcal{P}(V^*) \times \mathcal{P}(V^*) \longrightarrow \mathcal{P}(V^*)$ (language union);
- $\cdot \in \mathcal{P}(V^*) \times \mathcal{P}(V^*) \longrightarrow \mathcal{P}(V^*)$ (language concatenation);
- $* \in \mathcal{P}(V^*) \longrightarrow \mathcal{P}(V^*)$ (Kleene closure);
- $+ \in \mathcal{P}(V^*) \longrightarrow \mathcal{P}(V^*)$ ($+$ closure), and
- $? \in \mathcal{P}(V^*) \longrightarrow \mathcal{P}(V^*)$ (union with $\{\epsilon\}$, see Definition A.14).

Each of these operators corresponds (in the obvious way) to the operators of signature Σ. □

Definition 3.18 (Language denoted by an *RE*): The function $\mathcal{L}_{RE}$ is the (unique) homomorphism from the Σ-term algebra of *RE*'s to the Σ-algebra of regular languages. Function $\mathcal{L}_{RE}$ maps regular expressions to the languages they denote. □

Definition 3.19 (Equivalence ($\doteq$) of *RE*'s): Two regular expressions, E_0 and E_1, are said to be equivalent (written $E_0 \doteq E_1$, note the dot above the $=$) if and only if they denote the same language. □

Definition 3.20 (The *nullable* Σ-algebra): We define the *nullable* Σ-algebra as follows:

- The carrier set is $\{true, false\}$.

- The constants are: *true*, *false*, and *false* (corresponding respectively to ϵ, $\emptyset$, and $a : a \in V$). Here the constant *false* corresponds to $\emptyset$ and to all $a \in V$. The operators are: $\vee$ (disjunction), $\wedge$ (conjunction), the constant function *true*, the identity function, and (again) the constant function *true* (corresponding respectively to $\cup$, $\cdot$, $*$, $+$, and ?). The operators corresponding to $*$ and to ? are interesting because they map their argument to the constant *true*.

We denote the (unique) homomorphism from RE to this Σ-algebra as *Null*. □

Property 3.21 (The *nullable* Σ-algebra):The homomorphism *Null* has the property that for all $E \in RE$

$$\epsilon \in \mathcal{L}_{RE}(E) \equiv \mathit{Null}(E)$$

□

Definition 3.22 (RE reversal): Regular expression reversal is given by the postfix (superscript) isomorphism $R \in RE \longrightarrow RE$

$$\begin{array}{rcll}
\epsilon^R & = & \epsilon & \\
\emptyset^R & = & \emptyset & \\
a^R & = & a & \quad\quad\quad\quad \text{(for } a \in V\text{)} \\
(E_0 \cup E_1)^R & = & (E_0^R) \cup (E_1^R) & \\
(E_0 \cdot E_1)^R & = & (E_1^R) \cdot (E_0^R) & \\
(E^*)^R & = & (E^R)^* & \\
(E^+)^R & = & (E^R)^+ & \\
(E^?)^R & = & (E^R)^? &
\end{array}$$

Function R satisfies the obvious property that

$$(\forall\ E : E \in RE : (E^R)^R = E \wedge (\mathcal{L}_{RE}(E^R))^R = \mathcal{L}_{RE}(E))$$

□

Remark 3.23: The property satisfied by regular expression reversal implies that $\mathcal{L}_{RE}$ is an example of a symmetrical function (according to Definition A.22). □

4 Constructions based on regular expression structure

A finite automaton construction is any function f, such that the following diagram commutes:

$$\begin{tikzcd}
RE \arrow[r, "f"] \arrow[dr, "\mathcal{L}_{RE}"'] & FA \arrow[d, "\mathcal{L}_{FA}"] \\
 & \mathcal{L}_{\mathrm{reg}}
\end{tikzcd}$$

In this section, we will be defining some Σ-algebras with $[FA]_{\cong}$ as the carrier set; the idea behind the above commuting diagram still holds in this case, as all isomorphic FA's accept the same language. The isomorphism class of an FA corresponding to a given regular expression is the image of the regular expression under the (unique) homomorphism from RE to the other Σ-algebras. Such a homomorphism is a FA construction. Thompson's construction is considered first, followed by a derivation of Berry and Sethi's, McNaughton, Yamada and Glushkov's, and Aho, Sethi, and Ullman's constructions. We also consider methods of efficiently implementing some of the constructions, and methods of constructing FA's from extended regular expressions (see Definition 4.53).

4.1 Thompson's construction

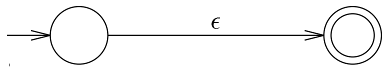
One Σ-algebra is based upon an RE to FA construction given by Thompson in [Thom68]. The explanations given in textbooks such as [HU79, Wood87, AU92, ASU86] are generally considered more readable than Thompson's original paper. None of those presentations made use of Σ-algebras.

Definition 4.1 (Thompson's Σ-algebra of FA's): The carrier set is $[FA]_{\cong}$. The operator requirement[3] is:

- For the binary operators, the representatives of the arguments must have disjoint state sets. For any two equivalence classes (under $\cong$) we can always choose a representative of each such that they satisfy this requirement.

The correctness of the operators[4] is not included here, but is discussed in Theorem B.1. Along with each operator we present a graphic representation of the operator. The operators are separated by horizontal lines for clarity. The operators (with subscript Th, for Thompson) are:

$$\begin{array}{rcll}
C_{\epsilon,Th} & = & \textbf{let} & q_0, q_1 \text{ be new states} \\
 & & \textbf{in} & \\
 & & & [(\{q_0,q_1\}, V, \emptyset, \{(q_0,q_1)\}, \{q_0\}, \{q_1\})]_{\cong} \\
 & & \textbf{end} &
\end{array}$$



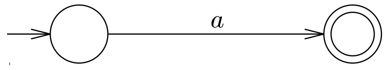
$$\begin{array}{rcll}
C_{\emptyset,Th} & = & \textbf{let} & q_0, q_1 \text{ be new states} \\
 & & \textbf{in} & \\
 & & & [(\{q_0,q_1\}, V, \emptyset, \emptyset, \{q_0\}, \{q_1\})]_{\cong} \\
 & & \textbf{end} &
\end{array}$$

[3] Σ-algebras presented in this section may have a list of items such as this, stating the requirements on the arguments for the correctness of the operators.

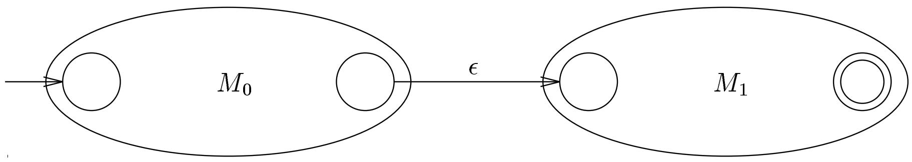
[4] For example, the concatenation operator is correct when (for all M_0, M_1 in Thompson's Σ-algebra) $\mathcal{L}_{FA}(C_{\cdot,Th}([M_0]_{\cong}, [M_1]_{\cong})) = \mathcal{L}_{FA}([M_0]_{\cong})\mathcal{L}_{FA}([M_1]_{\cong})$.

$$
\begin{array}{lcll}
C_{a,Th} & = & \textbf{let} & q_0, q_1 \text{ be new states} \\
 & & \textbf{in} & \\
 & & & [(\{q_0,q_1\}, V, \{(q_0,a,q_1)\}, \emptyset, \{q_0\}, \{q_1\})]_{\cong} \\
 & & \textbf{end} &
\end{array}
$$

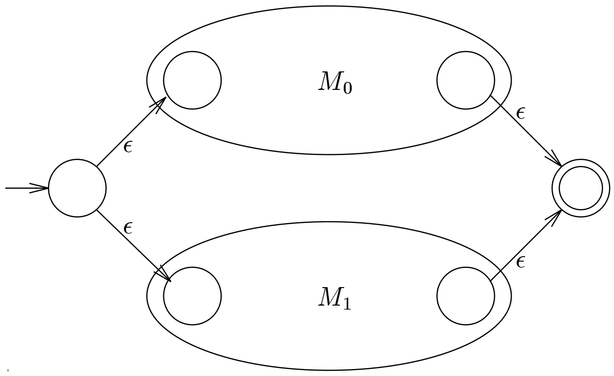
for all $a \in V$.



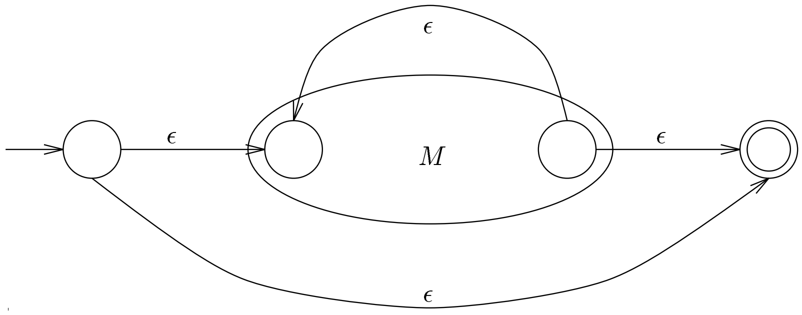
$$
\begin{array}{lcll}
C_{\cdot,Th}([M_0]_{\cong}, [M_1]_{\cong}) & = & \textbf{let} & (Q_0, V, T_0, E_0, S_0, F_0) = M_0 \\
 & & & (Q_1, V, T_1, E_1, S_1, F_1) = M_1 \\
 & & \textbf{in} & \\
 & & & \begin{array}{ll} \textbf{let} & E' = E_0 \cup E_1 \cup (F_0 \times S_1) \\ \textbf{in} & \\ & [(Q_0 \cup Q_1, V, T_0 \cup T_1, E', S_0, F_1)]_{\cong} \\ \textbf{end} & \end{array} \\
 & & \textbf{end} &
\end{array}
$$



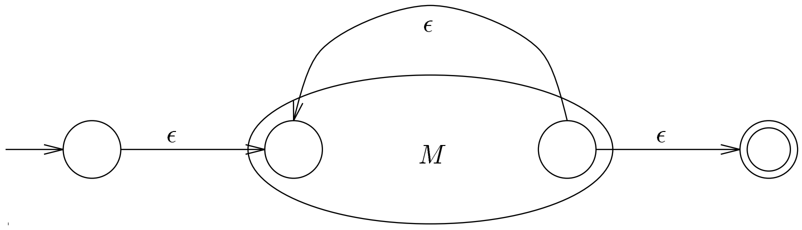
$$
\begin{array}{lcll}
C_{\cup,Th}([M_0]_{\cong}, [M_1]_{\cong}) & = & \textbf{let} & (Q_0, V, T_0, E_0, S_0, F_0) = M_0 \\
 & & & (Q_1, V, T_1, E_1, S_1, F_1) = M_1 \\
 & & & q_0, q_1 \text{ be new states} \\
 & & \textbf{in} & \\
 & & & \begin{array}{ll} \textbf{let} & Q' = Q_0 \cup Q_1 \cup \{q_0, q_1\} \\ & E' = E_0 \cup E_1 \cup (\{q_0\} \times (S_0 \cup S_1)) \\ & \quad \cup ((F_0 \cup F_1) \times \{q_1\}) \\ \textbf{in} & \\ & [(Q', V, T_0 \cup T_1, E', \{q_0\}, \{q_1\})]_{\cong} \\ \textbf{end} & \end{array} \\
 & & \textbf{end} &
\end{array}
$$



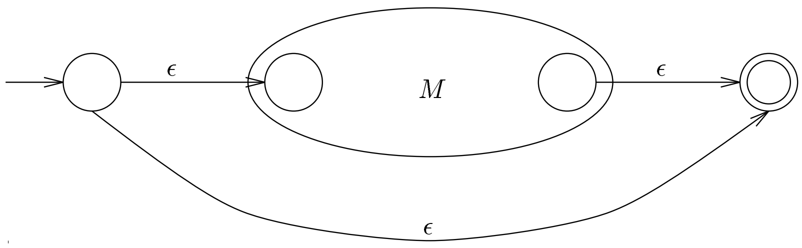
$$
\begin{array}{lcl}
C_{*,Th}([M]_{\cong}) & = & \textbf{let} \quad (Q,V,T,E,S,F) = M \\
& & \qquad\ \ q_0, q_1 \text{ be new states} \\
& & \textbf{in} \\
& & \qquad \textbf{let} \quad Q' = Q \cup \{q_0, q_1\} \\
& & \qquad\qquad\ \ E' = E \cup (\{q_0\} \times S) \cup (F \times S) \cup (F \times \{q_1\}) \cup \{(q_0, q_1)\} \\
& & \qquad \textbf{in} \\
& & \qquad\qquad\ \ [(Q', V, T, E', \{q_0\}, \{q_1\})]_{\cong} \\
& & \qquad \textbf{end} \\
& & \textbf{end}
\end{array}
$$



$$
\begin{array}{lcl}
C_{+,Th}([M]_{\cong}) & = & \textbf{let} \quad (Q,V,T,E,S,F) = M \\
& & \qquad\ \ q_0, q_1 \text{ be new states} \\
& & \textbf{in} \\
& & \qquad \textbf{let} \quad Q' = Q \cup \{q_0, q_1\} \\
& & \qquad\qquad\ \ E' = E \cup (\{q_0\} \times S) \cup (F \times S) \cup (F \times \{q_1\}) \\
& & \qquad \textbf{in} \\
& & \qquad\qquad\ \ [(Q', V, T, E', \{q_0\}, \{q_1\})]_{\cong} \\
& & \qquad \textbf{end} \\
& & \textbf{end}
\end{array}
$$



$$
\begin{array}{lcl}
C_{?,Th}([M]_{\cong}) & = & \textbf{let} \quad (Q,V,T,E,S,F) = M \\
& & \qquad\ \ q_0, q_1 \text{ be new states} \\
& & \textbf{in} \\
& & \qquad \textbf{let} \quad Q' = Q \cup \{q_0, q_1\} \\
& & \qquad\qquad\ \ E' = E \cup (\{q_0\} \times S) \cup (F \times \{q_1\}) \cup \{(q_0, q_1)\} \\
& & \qquad \textbf{in} \\
& & \qquad\qquad\ \ [(Q', V, T, E', \{q_0\}, \{q_1\})]_{\cong} \\
& & \qquad \textbf{end} \\
& & \textbf{end}
\end{array}
$$



These operators are symmetrical (see Definition A.22 for a definition of symmetrical operators and functions). Furthermore, they do not depend upon the choice of representative of the equivalence classes (under $\cong$). An automaton in Thompson's Σ-algebra (here we speak of a representative *FA*, instead of the isomorphism class) has the following properties:

- It has a single start state with no in-transitions.
- It has a single final state with no out-transitions.
- Every state has either a single in-transition on a symbol (in V), or at most two ϵ in-transitions.
- Every state has either a single out-transition on a symbol (in V), or at most two ϵ out-transitions.

These properties are symmetrical because the operators are symmetrical. Hopcroft and Ullman have shown [HU79] that in practice these properties facilitate the quick simulation of M. For the remainder of this paper we will not duplicate properties such as these, but rather state whether the operator is symmetrical. □

Remark 4.2: In the literature, these operators are usually presented as having arguments and results of type *FA* instead of $[FA]_{\cong}$. Such a presentation is given in terms of particular representatives, and ignores the nondeterminism in choosing new states. □

Construction 4.3 (Thompson): Thompson's construction is the (unique) homomorphism *Th* from *RE* to Thompson's Σ-algebra of *FA*'s. □

Example 4.4 (Thompson's construction): We construct a particular representative[5] of

$$\begin{aligned} Th((a \cup \epsilon)b^*) &= C_{\cdot,Th}(Th(a \cup \epsilon), Th(b^*)) \\ &= C_{\cdot,Th}(C_{\cup,Th}(Th(a), Th(\epsilon)), C_{*,Th}(b)) \\ &= C_{\cdot,Th}(C_{\cup,Th}(C_{a,Th}, C_{\epsilon,Th}), C_{*,Th}(C_{b,Th})) \end{aligned}$$

(The regular expression is taken from Example 3.15.) The representative is shown in Figure 2. □

In the next two subsections, we consider two algorithms that construct an *FA* (from a regular expression) based on the top-down syntactic structure of the regular expression. In these two constructions, we use regular expressions as syntactic objects denoting regular languages.

The first construction is a top-down version of Thompson's construction. The second one is also top-down, but constructs a so-called ϵ-lookahead automaton. Such an automaton can be efficiently simulated or it can be converted to an efficient program, accepting the language of the automaton.

---

[5] Obviously, constructing the entire equivalence class of isomorphic *FA*'s is not possible

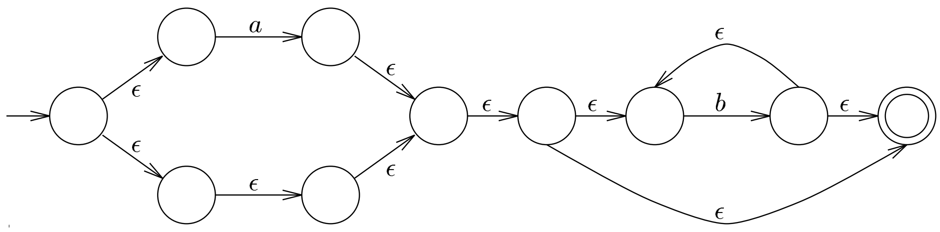


Figure 2: A representative *FA* of the isomorphism class $Th((a \cup \epsilon)b^*)$.

4.1.1 A top-down version of Thompson's construction

The top-down version of Thompson's construction is a practical implementation of homomorphism *Th*. It is a function of three parameters: a start state s, a regular expression E, and a final state f. It produces an *FA*, with start state s and final state f, accepting the language $\mathcal{L}_{RE}(E)$.

Construction 4.5 (Top-down Thompson's): We assume a universe of available states U, to define function

$$td \in U \times RE \times U \longrightarrow FA$$

The function is defined recursively on the structure of regular expressions:

$$\begin{array}{lcll} td(s,\epsilon,f) & = & (\{s,f\},V,\emptyset,\{(s,f)\},\{s\},\{f\}) & \\ td(s,\emptyset,f) & = & (\{s,f\},V,\emptyset,\emptyset,\{s\},\{f\}) & \\ td(s,a,f) & = & (\{s,f\},V,\{(s,a,f)\},\emptyset,\{s\},\{f\}) & \text{(for all } a \in V) \end{array}$$

$$\begin{array}{lcl} td(s,E_0 \cdot E_1,f) & = & \textbf{let}\quad p,q \text{ be new states} \\ & & \textbf{in} \\ & & \qquad \textbf{let}\quad (Q_0,V,T_0,E_0,\{s\},\{p\}) = td(s,E_0,p) \\ & & \qquad \qquad (Q_1,V,T_1,E_1,\{q\},\{f\}) = td(q,E_1,f) \\ & & \qquad \textbf{in} \\ & & \qquad \qquad (Q_0 \cup Q_1,V,T_0 \cup T_1,E_0 \cup E_1 \cup \{(p,q)\},\{s\},\{f\}) \\ & & \qquad \textbf{end} \\ & & \textbf{end} \end{array}$$

$$\begin{array}{lcl} td(s,E_0 \cup E_1,f) & = & \textbf{let}\quad p,q,r,t \text{ be new states} \\ & & \textbf{in} \\ & & \qquad \textbf{let}\quad (Q_0,V,T_0,E_0,\{p\},\{q\}) = td(p,E_0,q) \\ & & \qquad \qquad (Q_1,V,T_1,E_1,\{r\},\{t\}) = td(r,E_1,t) \\ & & \qquad \textbf{in} \\ & & \qquad \qquad (Q_0 \cup Q_1 \cup \{s,f\},V,T_0 \cup T_1,E_0 \cup E_1 \\ & & \qquad \qquad \quad \cup (\{s\} \times \{p,r\}) \cup (\{q,t\} \times \{f\}),\{s\},\{f\}) \\ & & \qquad \textbf{end} \\ & & \textbf{end} \end{array}$$

$$\begin{array}{lcl} td(s,E^*,f) & = & \textbf{let}\quad p,q \text{ be new states} \\ & & \textbf{in} \\ & & \qquad \textbf{let}\quad (Q,V,T,E,\{p\},\{q\}) = td(p,E,q) \\ & & \qquad \textbf{in} \\ & & \qquad \qquad (Q \cup \{s,f\},V,T,E \cup \{(s,p),(q,p),(q,f),(s,f)\},\{s\},\{f\}) \\ & & \qquad \textbf{end} \\ & & \textbf{end} \end{array}$$

$$
\begin{array}{lcl}
td(s, E^+, f) & = & \textbf{let} \quad p, q \text{ be new states} \\
& & \textbf{in} \\
& & \qquad \textbf{let} \quad (Q, V, T, E, \{p\}, \{q\}) = td(p, E, q) \\
& & \qquad \textbf{in} \\
& & \qquad\qquad (Q \cup \{s, f\}, V, T, E \cup \{(s, p), (q, p), (q, f)\}, \{s\}, \{f\}) \\
& & \qquad \textbf{end} \\
& & \textbf{end}
\end{array}
$$

$$
\begin{array}{lcl}
td(s, E^?, f) & = & \textbf{let} \quad p, q \text{ be new states} \\
& & \textbf{in} \\
& & \qquad \textbf{let} \quad (Q, V, T, E, \{p\}, \{q\}) = td(p, E, q) \\
& & \qquad \textbf{in} \\
& & \qquad\qquad (Q \cup \{s, f\}, V, T, E \cup \{(s, p), (q, f), (s, f)\}, \{s\}, \{f\}) \\
& & \qquad \textbf{end} \\
& & \textbf{end}
\end{array}
$$

Function *td* satsifies the property that, for all $E \in RE$:

$$
\begin{array}{lcl}
Th(E) & = & \textbf{let} \quad s, f \text{ be new states} \\
& & \textbf{in} \\
& & \qquad [td(s, E, f)]_{\cong} \\
& & \textbf{end}
\end{array}
$$

□

The advantage of function *td* over homomorphism *Th* (Construction 4.3) is one of implementation. In Thompson's construction, the subparts of the final *FA* are constructed in isolation; when two subparts are combined some states may have to be renamed to ensure that the subparts have disjoint state sets. In the top-down construction, more global knowledge is available concerning the final *FA* and this type of problem is avoided. (In practice, function *td* would make use of a global variable: the set of remaining available states.)

We do not prove the correctness of construction *td* in this paper.

4.1.2 Constructing ϵ-lookahead automata

In this subsection, we extend the top-down Thompson construction (function *td*) to construct ϵ-lookahead finite automata (*LAFA*). In an *LAFA*, every ϵ-transition is qualified by a symbol of V (known as the lookahead symbol). When simulating an *LAFA*, an ϵ-transition can be taken if the next symbol of the input string matches the lookahead symbol of the ϵ-transition. Naturally, for any given state, it is desirable that there only be one ϵ-transition from the state on any given symbol. The following definitions formalize this.

Definition 4.6 (ϵ-lookahead automata): An ϵ-lookahead finite automaton (*LAFA*) is a 6-tuple (Q, V, T, E, S, F) which is a normal *FA* with one exception:

- ϵ-transition relation is now $E \in \mathcal{P}(Q \times V \times Q)$ instead of $E \in \mathcal{P}(Q \times Q)$.

□

Remark 4.7: A more commonly presented definition of *LAFA*'s involves both ϵ-lookahead and normal ϵ-transitions (also called *don't-care transitions*). Since we have combined the two, we implement normal ϵ-transitions as lookahead transitions, where the lookahead set is V (the entire alphabet). □

Remark 4.8: Naturally, we extend such functions as $\overleftarrow{\mathcal{L}}$, $\overrightarrow{\mathcal{L}}$, and $\mathcal{L}_{FA}$ to use the definition of a *LAFA*. As a result, the language accepted by an *LAFA* is in accordance with the intuitive interpretation of an *LAFA*. □

Remark 4.9: In order to present the following definition, we require the definition of function $First \in RE \longrightarrow \mathcal{P}(V)$. Function *First* is defined in Definition 4.60. Informally, $First(E)$ is the set of all alphabet symbols that can occur as the first symbol of a word in $\mathcal{L}_{RE}(E)$. □

Definition 4.10 (Lookahead function): In order to make the definition of the *LAFA* construction readable, we introduce function $look \in RE \times \mathcal{P}(V) \longrightarrow \mathcal{P}(V)$, defined as:

$$look(E, L) = First(E) \cup \textbf{if } (Null(E)) \textbf{ then } L \textbf{ else } \emptyset \textbf{ fi}$$

Argument L is called the set of *follow symbols.* □

We now define an *LAFA* construction, based on the top-down version of Thompson's construction.

Construction 4.11 (ϵ-lookahead finite automaton): We define function K which takes four parameters: a start state s, a regular expression, final state f, and a lookahead set $L \in \mathcal{P}(V)$. As with the top-down version of Thompson's construction, we assume a universe of states U. Function $K \in U \times RE \times U \times \mathcal{P}(V) \longrightarrow LAFA$ is defined recursively on the structure of regular expressions:

$$\begin{array}{lcll}
K(s,\epsilon,f,L) & = & (\{s,f\},V,\emptyset,\{s\}\times L\times\{f\},\{s\},\{f\}) & \\
K(s,\emptyset,f,L) & = & (\{s,f\},V,\emptyset,\emptyset,\{s\},\{f\}) & \\
K(s,a,f,L) & = & (\{s,f\},V,\{(s,a,f)\},\emptyset,\{s\},\{f\}) & \text{(for all } a \in V\text{)}
\end{array}$$

$$\begin{array}{lcl}
K(s, E_0 \cdot E_1, f, L) & = & \textbf{let} \quad p, q \text{ be new states} \\
& & \textbf{in} \\
& & \qquad \textbf{let} \quad (Q_0, V, T_0, E_0, \{s\}, \{p\}) = K(s, E_0, p, look(E_1, L)) \\
& & \qquad \qquad\;\; (Q_1, V, T_1, E_1, \{q\}, \{f\}) = K(q, E_1, f, L) \\
& & \qquad \textbf{in} \\
& & \qquad \qquad (Q_0 \cup Q_1, V, T_0 \cup T_1, E_0 \cup E_1 \\
& & \qquad \qquad \quad \cup (\{p\} \times look(E_1, L) \times \{q\}), \{s\}, \{f\}) \\
& & \qquad \textbf{end} \\
& & \textbf{end}
\end{array}$$

$$\begin{array}{lcl}
K(s, E_0 \cup E_1, f, L) & = & \textbf{let} \quad p, q, r, t \text{ be new states} \\
& & \textbf{in} \\
& & \qquad \textbf{let} \quad (Q_0, V, T_0, E_0, \{p\}, \{q\}) = K(p, E_0, q, L) \\
& & \qquad \qquad\;\; (Q_1, V, T_1, E_1, \{r\}, \{t\}) = K(r, E_1, t, L) \\
& & \qquad \textbf{in} \\
& & \qquad \qquad (Q_0 \cup Q_1 \cup \{s, f\}, V, T_0 \cup T_1, E_0 \cup E_1 \\
& & \qquad \qquad \quad \cup (\{s\} \times look(E_0, L) \times \{p\}) \\
& & \qquad \qquad \quad \cup (\{s\} \times look(E_1, L) \times \{r\}) \\
& & \qquad \qquad \quad \cup (\{q, t\} \times L \times \{f\}), \{s\}, \{f\}) \\
& & \qquad \textbf{end} \\
& & \textbf{end}
\end{array}$$

$$\begin{array}{lcl}
K(s, E^*, f, L) & = & \textbf{let} \quad p, q \text{ be new states} \\
& & \textbf{in} \\
& & \qquad \textbf{let} \quad (Q, V, T, E, \{p\}, \{q\}) = K(p, E, q, L \cup First(E)) \\
& & \qquad \textbf{in} \\
& & \qquad \qquad (Q \cup \{s, f\}, V, T, E \cup (\{s, q\} \times First(E) \times \{p\}) \\
& & \qquad \qquad \quad \cup (\{s, q\} \times L \times \{f\}), \{s\}, \{f\}) \\
& & \qquad \textbf{end} \\
& & \textbf{end}
\end{array}$$

$$
\begin{array}{lll}
K(s, E^+, f, L) & = & \textbf{let} \quad p, q \text{ be new states} \\
& & \textbf{in} \\
& & \quad \textbf{let} \quad (Q, V, T, E, \{p\}, \{q\}) = K(p, E, q, L \cup \mathit{First}(E)) \\
& & \quad \textbf{in} \\
& & \quad\quad (Q \cup \{s, f\}, V, T, E \cup (\{s, q\} \times \mathit{First}(E) \times \{p\}) \\
& & \quad\quad\quad \cup (\{q\} \times L \times \{f\}), \{s\}, \{f\}) \\
& & \quad \textbf{end} \\
& & \textbf{end}
\end{array}
$$

$$
\begin{array}{lll}
K(s, E^?, f, L) & = & \textbf{let} \quad p, q \text{ be new states} \\
& & \textbf{in} \\
& & \quad \textbf{let} \quad (Q, V, T, E, \{p\}, \{q\}) = K(p, E, q, L) \\
& & \quad \textbf{in} \\
& & \quad\quad (Q \cup \{s, f\}, V, T, E \cup (\{s\} \times \mathit{First}(E) \times \{p\}) \\
& & \quad\quad\quad \cup (\{s, q\} \times L \times \{f\}), \{s\}, \{f\}) \\
& & \quad \textbf{end} \\
& & \textbf{end}
\end{array}
$$

□

Remark 4.12: Since we make use of a single symbol of lookahead, we assume that the input string always has an end-marker \$ concatenated on its right. We assume that $\$ \in V$ and that \$ does not appear elsewhere in the regular expression. This means that, for $E \in RE$:

$$
\begin{array}{l}
\textbf{let} \quad s, f \text{ be new states} \\
\textbf{in} \\
\quad K(s, E, f, \{\$\}) \\
\textbf{end}
\end{array}
$$

is a *LAFA* accepting $\mathcal{L}_{RE}(E)$. □

Definition 4.13 (Deterministic *LAFA*'s): A *LAFA* is deterministic if and only if it has at most one start state and no state has more than one out-transition (either an ϵ-lookahead or a normal transition) on any given alphabet symbol. □

We present some determinism conditions that ensure that Construction 4.11 produces deterministic *LAFA*'s.

Definition 4.14 (Determinism conditions): In order for function K to produce a deterministic *LAFA* we impose the following requirements for particular cases of K:

- For $K(s, E_0 \cup E_1, f, L)$ we require that $\mathit{look}(E_0, L) \cap \mathit{look}(E_1, L) = \emptyset$.
- For $K(s, E^*, f, L)$, $K(s, E^+, f, L)$, and $K(s, E^?, f, L)$ we require that $\mathit{First}(E) \cap L = \emptyset$.

□

Remark 4.15: The lookahead transitions in *LAFA*'s make them are more efficient to simulate than an equivalent *FA* constructed with Thompson's construction. Simulation of a deterministic *LAFA* is as efficient as the simulation of a *DFA*. □

Example 4.16 (*LAFA*): Given new states s, f, we construct the deterministic *LAFA* $K(s, (a \cup \epsilon)b^*, f, \{\$\})$. The ϵ-lookahead transitions are labeled with both ϵ and the lookahead symbols. The state graph is given in Figure 3. □

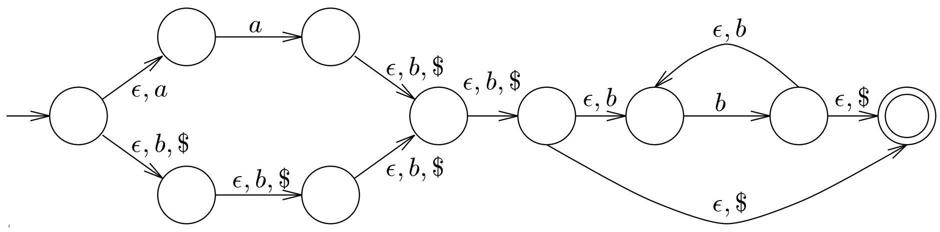


Figure 3: The *LAFA* $K((a \cup \epsilon)b^*)$.

Construction 4.17 (Creating a program from a *LAFA*): A deterministic *LAFA* can also be converted into a program which is a hard-coded simulation of the *LAFA*. We now describe a mapping $N \in RE \times \mathcal{P}(V) \longrightarrow GCL$, where *GCL* denotes the set of all guarded commands programs [Dijk76]. This construction is based upon the *LAFA* construction. The created programs are correct when the determinism conditions of Definition 4.14 hold. In the generated program, we assume that variable $w \in V^*$ is the input string (with an end-marker \$ concatenated on its right), and that $hd(w)$ refers to the first symbol of w and $tl(w)$ refers to the remainder of w. We annotate the program fragments (in the definition of N) with the state names (in braces) in the corresponding definition of Construction 4.11. (The semantics of the guarded commands specify that if none of the guards in an **if-fi** statement are true, the statement is equivalent to **abort**.)

$$
\begin{array}{lcll}
N(\epsilon, L) & = & \{s\} & \\
 & & \textbf{if } hd(w) \in L \longrightarrow \textbf{ skip} & \\
 & & \textbf{fi} & \\
 & & \{f\} & \\
N(\emptyset, L) & = & \{s\} \textbf{ abort}\{f\} & \\
N(a, L) & = & \{s\} & \text{(for all } a \in V) \\
 & & \textbf{if } hd(w) = a \longrightarrow w := tl(w) & \\
 & & \textbf{fi} & \\
 & & \{f\} & \\
N(E_0 \cdot E_1, L) & = & \{s\}N(E_0, look(E_1, L))\{p\}; & \\
 & & \{q\}N(E_1, L)\{f\} & \\
N(E_0 \cup E_1, L) & = & \{s\} & \\
 & & \textbf{if } hd(w) \in look(E_0, L) \longrightarrow \{p\}N(E_0, L)\{q\} & \\
 & & [\!]\ hd(w) \in look(E_1, L) \longrightarrow \{r\}N(E_1, L)\{t\} & \\
 & & \textbf{fi} & \\
 & & \{f\} & \\
N(E^*, L) & = & \{s\} & \\
 & & \textbf{do } hd(w) \in First(E) \longrightarrow \{p\}N(E, First(E) \cup L)\{q\} & \\
 & & \textbf{od} & \\
 & & \{f\} & \\
N(E^+, L) & = & \{s\} & \\
 & & \textbf{repeat } \{p\}N(E, First(E) \cup L)\{q\} & \\
 & & \textbf{until } hd(w) \notin First(E) & \\
 & & \{f\} & \\
N(E^?, L) & = & \{s\} & \\
 & & \textbf{if } hd(w) \in First(E) \longrightarrow \{p\}N(E, L)\{q\} & \\
 & & [\!]\ hd(w) \in L \longrightarrow \textbf{ skip} & \\
 & & \textbf{fi} & \\
 & & \{f\} &
\end{array}
$$

As with Construction 4.11 we concatenate an end-marker \$ on the right of the input string w. The entire program of the acceptor (for $E \in RE$) is:

$\{w \in V^*\{\$\}\}$
$N(E, \{\$\})$;
if $w \neq \$ \longrightarrow$ **abort**
fi
$\{w \in \mathcal{L}_{RE}(E)\}$

Termination of the program is equivalent to $w \in \mathcal{L}_{RE}(E)$. □

Example 4.18 (Programs from *LAFA*'s): We construct the program corresponding to $(a \cup \epsilon)b^*$.

$\{w \in V^*\{\$\}\}$
if $hd(w) \in \{a\} \longrightarrow$
  if $hd(w) = a \longrightarrow w := tl(w)$
  fi
$[\!]$ $hd(w) \in \{b, \$\} \longrightarrow$
  if $hd(w) \in \{b, \$\} \longrightarrow$ **skip**
  fi
fi;
do $hd(w) \in \{b\} \longrightarrow$
  if $hd(w) = b \longrightarrow w := tl(w)$
  fi
od
if $w \neq \$ \longrightarrow$ **abort**
fi
$\{w \in \{a, \epsilon\}\{b\}^*\}$

□

4.2 Towards the Berry-Sethi construction

We now consider Σ-algebras of ϵ-*free* *FA*'s. One such Σ-algebra can be given with symmetrical operators.

Definition 4.19 (Symmetrical ϵ-*free* Σ-algebra operators): The carrier set is $[FA]_{\cong}$. The operator requirement is (as with Thompson's Σ-algebra):

- For the binary operators, the representatives of the arguments must have disjoint state sets.

The symmetrical ϵ-*free* preserving operators of the Σ-algebra are defined using Thompson's Σ-algebra operators and symmetrical function $remove_{\epsilon,sym}$ (which is extended to $[FA]_{\cong} \longrightarrow [FA]_{\cong}$):

$$\begin{array}{lcll}
C_{\epsilon,sym} & = & remove_{\epsilon,sym} \circ C_{\epsilon,Th} & \\
C_{\emptyset,sym} & = & remove_{\epsilon,sym} \circ C_{\emptyset,Th} & \\
C_{a,sym} & = & remove_{\epsilon,sym} \circ C_{a,Th} & \text{(for all } a \in V) \\
C_{\cdot,sym} & = & remove_{\epsilon,sym} \circ C_{\cdot,Th} & \\
C_{\cup,sym} & = & remove_{\epsilon,sym} \circ C_{\cup,Th} & \\
C_{*,sym} & = & remove_{\epsilon,sym} \circ C_{*,Th} & \\
C_{+,sym} & = & remove_{\epsilon,sym} \circ C_{+,Th} & \\
C_{?,sym} & = & remove_{\epsilon,sym} \circ C_{?,Th} &
\end{array}$$

These operators are symmetrical since they are compositions of symmetrical operators (see Proposition A.23). An *FA* in this Σ-algebra has the property that it is ϵ-*free*. □

These operators are cumbersome to present fully. Furthermore, they are not particularly useful in practice. For this reason, we now consider asymmetrically defined ϵ-*free* preserving operators.

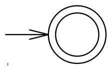
The first asymmetrical ϵ-*free* preserving Σ-algebra operators that we consider are the left-biased ones. The image of a *RE* in this Σ-algebra is easier to compute than its image in the Σ-algebra given in Definition 4.19).

Definition 4.20 (Σ-algebra of left-biased ϵ-*free* operators): The carrier set is $[FA]_{\cong}$. The operator requirements are:

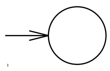
- For binary operators, the representatives of the arguments must have disjoint state sets.
- The following is required of the representatives of each argument:
  - it is ϵ-*free*,
  - it has a single start state, and
  - the single start state has no in-transitions.

A proof of the correctness of these operators is outlined in Theorem B.2. As in Thompson's Σ-algebra, each operator is presented here with a graphic representation of the operator[6]. Parts of the operator definitions are intentionally clumsy or verbose. This is done to facilitate the derivation of a Σ-algebra of reduced *FA*'s (in Definition 4.29). The operators are:

$$
\begin{array}{lcll}
C_{\epsilon,LBFA} & = & \textbf{let} & q_0 \text{ be a new state} \\
 & & \textbf{in} & \\
 & & & [(\{q_0\}, V, \emptyset, \emptyset, \{q_0\}, \{q_0\})]_{\cong} \\
 & & \textbf{end} &
\end{array}
$$

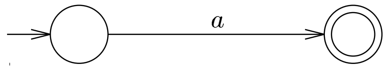


$$
\begin{array}{lcll}
C_{\emptyset,LBFA} & = & \textbf{let} & q_0 \text{ be a new state} \\
 & & \textbf{in} & \\
 & & & [(\{q_0\}, V, \emptyset, \emptyset, \{q_0\}, \emptyset)]_{\cong} \\
 & & \textbf{end} &
\end{array}
$$



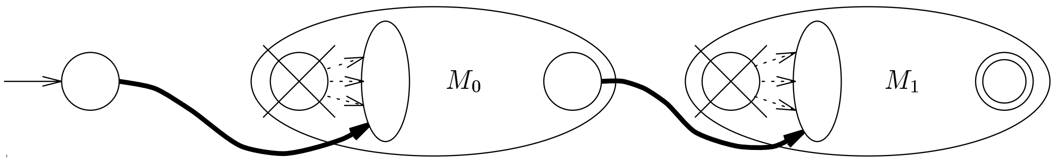
$$
\begin{array}{lcll}
C_{a,LBFA} & = & \textbf{let} & q_0, q_1 \text{ be new states} \\
 & & \textbf{in} & \\
 & & & [(\{q_0, q_1\}, V, \{(q_0, a, q_1)\}, \emptyset, \{q_0\}, \{q_1\})]_{\cong} \\
 & & \textbf{end} &
\end{array}
$$

for all $a \in V$.

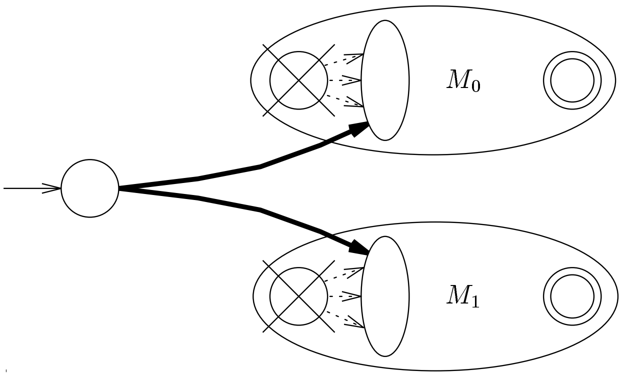


[6]The graphic representations of the operators depict only the simplest cases of each operator. Thick arrowed lines are intended to depict multiple transitions, while dotted arrowed lines are transitions that are removed from the constructed *FA*. In the case of the non-constant operators, the start states (of the arguments) is struck out indicating that it is removed.

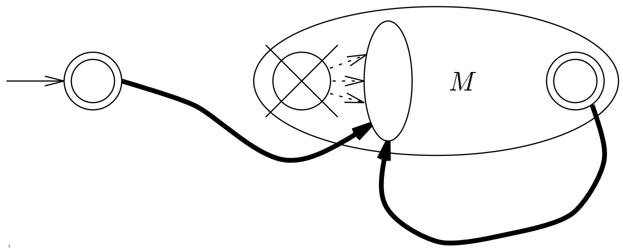
$$
\begin{array}{lcl}
C_{\cdot,LBFA}([M_0]_{\cong},[M_1]_{\cong}) & = & \textbf{let} \quad (Q_0,V,T_0,\emptyset,\{s_0\},F_0)=M_0 \\
& & \phantom{\textbf{let}} \quad (Q_1,V,T_1,\emptyset,\{s_1\},F_1)=M_1 \\
& & \phantom{\textbf{let}} \quad N_0=\epsilon\in\mathcal{L}_{FA}(M_0) \\
& & \phantom{\textbf{let}} \quad N_1=\epsilon\in\mathcal{L}_{FA}(M_1) \\
& & \phantom{\textbf{let}} \quad N=\epsilon\in(\mathcal{L}_{FA}(M_0)\mathcal{L}_{FA}(M_1)) \\
& & \phantom{\textbf{let}} \quad q_0 \text{ be a new state} \\
& & \textbf{in} \\
& & \quad \textbf{let} \quad Q'=Q_0\setminus\{s_0\}\cup Q_1\setminus\{s_1\}\cup\{q_0\} \\
& & \quad \phantom{\textbf{let}} \quad T'=T_0\cup T_1\cup(F_0\times T_1(s_1)) \\
& & \quad \phantom{\textbf{let}} \quad \qquad \cup(\{q_0\}\times(T_0(s_0) \\
& & \quad \phantom{\textbf{let}} \quad \qquad \cup\,\textbf{if}\ (N_0)\ \textbf{then}\ T_1(s_1)\ \textbf{else}\ \emptyset\ \textbf{fi})) \\
& & \quad \phantom{\textbf{let}} \quad F'=F_1\cup\textbf{if}\ (N_1)\ \textbf{then}\ F_0\ \textbf{else}\ \emptyset\ \textbf{fi} \\
& & \quad \phantom{\textbf{let}} \quad \qquad \cup\,\textbf{if}\ (N)\ \textbf{then}\ \{q_0\}\ \textbf{else}\ \emptyset\ \textbf{fi} \\
& & \quad \textbf{in} \\
& & \quad \phantom{\textbf{let}} \quad [(Q',V,T'\cap(Q'\times V\times Q'), \\
& & \quad \phantom{\textbf{let}} \quad \qquad \emptyset,\{q_0\},F'\cap Q')]_{\cong} \\
& & \quad \textbf{end} \\
& & \textbf{end}
\end{array}
$$



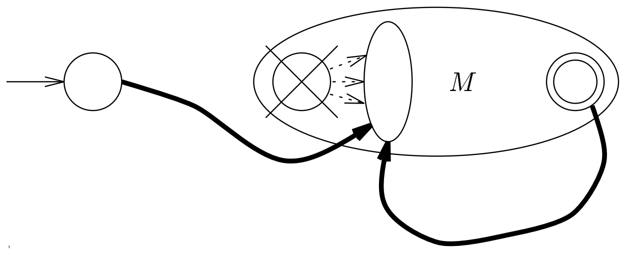
$$
\begin{array}{lcl}
C_{\cup,LBFA}([M_0]_{\cong},[M_1]_{\cong}) & = & \textbf{let} \quad (Q_0,V,T_0,\emptyset,\{s_0\},F_0)=M_0 \\
& & \phantom{\textbf{let}} \quad (Q_1,V,T_1,\emptyset,\{s_1\},F_1)=M_1 \\
& & \phantom{\textbf{let}} \quad N=\epsilon\in(\mathcal{L}_{FA}(M_0)\cup\mathcal{L}_{FA}(M_1)) \\
& & \phantom{\textbf{let}} \quad q_0 \text{ be a new state} \\
& & \textbf{in} \\
& & \quad \textbf{let} \quad Q'=Q_0\setminus\{s_0\}\cup Q_1\setminus\{s_1\}\cup\{q_0\} \\
& & \quad \phantom{\textbf{let}} \quad T'=T_0\cup T_1\cup(\{q_0\}\times(T_0(s_0)\cup T_1(s_1))) \\
& & \quad \phantom{\textbf{let}} \quad F'=F_0\cup F_1\cup\textbf{if}\ (N)\ \textbf{then}\ \{q_0\}\ \textbf{else}\ \emptyset\ \textbf{fi} \\
& & \quad \textbf{in} \\
& & \quad \phantom{\textbf{let}} \quad [(Q',V,T'\cap(Q'\times V\times Q'), \\
& & \quad \phantom{\textbf{let}} \quad \qquad \emptyset,\{q_0\},F'\cap Q')]_{\cong} \\
& & \quad \textbf{end} \\
& & \textbf{end}
\end{array}
$$



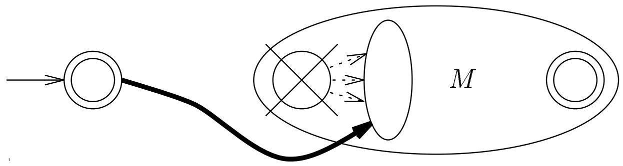
$$
\begin{array}{lcll}
C_{*,LBFA}([M]_{\cong}) & = & \textbf{let} & (Q,V,T,\emptyset,\{s\},F) = M \\
 & & & N = \epsilon \in \mathcal{L}_{FA}(M)^* \qquad\qquad \text{(see Remark 4.21)} \\
 & & & q_0 \text{ be a new state} \\
 & & \textbf{in} & \\
 & & & \quad \textbf{let} \quad Q' = Q \setminus \{s\} \cup \{q_0\} \\
 & & & \quad \phantom{\textbf{let}} \quad T' = T \cup (F \cup \{q_0\}) \times T(s) \\
 & & & \quad \phantom{\textbf{let}} \quad F' = F \cup \textbf{if } (N) \textbf{ then } \{q_0\} \textbf{ else } \emptyset \textbf{ fi} \\
 & & & \quad \textbf{in} \\
 & & & \quad \phantom{\textbf{let}} \quad [(Q',V,T' \cap (Q' \times V \times Q'), \\
 & & & \quad \phantom{\textbf{let}} \quad \quad \emptyset,\{q_0\},F' \cap Q')]_{\cong} \\
 & & & \quad \textbf{end} \\
 & & \textbf{end} &
\end{array}
$$



$$
\begin{array}{lcll}
C_{+,LBFA}([M]_{\cong}) & = & \textbf{let} & (Q,V,T,\emptyset,\{s\},F) = M \\
 & & & N = \epsilon \in \mathcal{L}_{FA}(M)^+ \\
 & & & q_0 \text{ be a new state} \\
 & & \textbf{in} & \\
 & & & \quad \textbf{let} \quad Q' = Q \setminus \{s\} \cup \{q_0\} \\
 & & & \quad \phantom{\textbf{let}} \quad T' = T \cup (F \cup \{q_0\}) \times T(s) \\
 & & & \quad \phantom{\textbf{let}} \quad F' = F \cup \textbf{if } (N) \textbf{ then } \{q_0\} \textbf{ else } \emptyset \textbf{ fi} \\
 & & & \quad \textbf{in} \\
 & & & \quad \phantom{\textbf{let}} \quad [(Q',V,T' \cap (Q' \times V \times Q'), \\
 & & & \quad \phantom{\textbf{let}} \quad \quad \emptyset,\{q_0\},F' \cap Q')]_{\cong} \\
 & & & \quad \textbf{end} \\
 & & \textbf{end} &
\end{array}
$$



$$
\begin{array}{lcll}
C_{?,LBFA}([M]_{\cong}) & = & \textbf{let} & (Q, V, T, \emptyset, \{s\}, F) = M \\
& & & N = \epsilon \in \mathcal{L}_{FA}(M)^{?} \qquad\qquad \text{(see Remark 4.21)} \\
& & & q_0 \text{ be a new state} \\
& & \textbf{in} & \\
& & & \textbf{let} \quad Q' = Q \setminus \{s\} \cup \{q_0\} \\
& & & \phantom{\textbf{let}} \quad T' = T \cup (\{q_0\} \times T(s)) \\
& & & \phantom{\textbf{let}} \quad F' = F \cup \textbf{if } (N) \textbf{ then } \{q_0\} \textbf{ else } \emptyset \textbf{ fi} \\
& & & \textbf{in} \\
& & & \phantom{\textbf{let}} \quad [(Q', V, T' \cap (Q' \times V \times Q'), \\
& & & \phantom{\textbf{let}} \qquad \emptyset, \{q_0\}, F' \cap Q')]_{\cong} \\
& & & \textbf{end} \\
& & \textbf{end} &
\end{array}
$$



The choice of representatives in these operators is irrelevant. For construction purposes, we note that $\epsilon \in \mathcal{L}_{FA}(M) \equiv s \in F$.

Let *LBFA* (where $LBFA \subset FA$) denote the set of all finite automata that are images[7] in this Σ-algebra of some $E \in RE$. (That is, *LBFA* is the smallest set that contains the *LBFA* constants and is closed under the *LBFA* operators.) An *LBFA* has the following properties:

- It is *ϵ-free*.
- It has a single start state.
- The single start state has no in-transitions.
- All in-transitions to a state are on the same symbol (in V). This can be seen by considering the constants $C_{a,LBFA}$ (for all $a \in V$), which are the only operators introducing new transitions on an alphabet symbol.

Only the constants are symmetrical. □

Remark 4.21: Parts of the operator definitions of Definition 4.20 are intentionally clumsy; they are presented this way to facilitate the derivation of a Σ-algebra of reduced *FA*'s (Definition 4.29). □

Construction 4.22 (Left-biased finite automata): Define construction $lbfa \in RE \longrightarrow [LBFA]_{\cong}$ to be the unique homomorphism from *RE*'s to $[LBFA]_{\cong}$. □

Example 4.23 (Σ-algebra of *LBFA*'s): We construct a representative of the isomorphism class $lbfa((a \cup \epsilon)b^*)$ (the regular expression is from Example 3.15). The representative is shown in Figure 4. □

Computing within the Σ-algebra of *LBFA*'s is inefficient. Each operator defined above does much redundant work. For example, the start states of the arguments to the operators are always removed, with only the out-transitions from the argument's start state being used. Additionally, the **if-fi** structures within the final states definition are of the same structure in each operator. We wish to introduce an encoding of *LBFA*'s that will allow us to find cheap constructions that are equivalent to *lbfa*. We now describe such an encoding.

A method of encoding an *LBFA* $(Q, V, T, \emptyset, \{s\}, F)$ is:

[7] The images are really elements of $[LBFA]_{\cong}$. We consider a particular representative of the image.

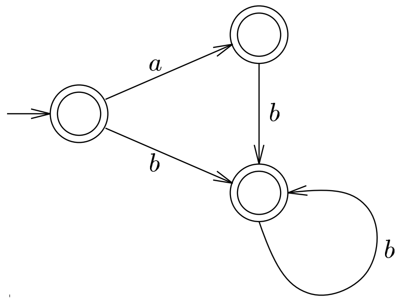


Figure 4: A representative *LBFA* of the isomorphism class $lbfa((a \cup \epsilon)b^*)$.

- The ϵ-transitions are not needed (since *LBFA*'s are ϵ-*free*).
- State s has no in-transitions; only $T(s)$ (the out-transitions from the start state) and $s \in F$ are needed.
- All in-transitions to a state are on the same symbol (in V). Therefore, a state-to-symbol map can be used, and the symbol components of T and $T(s)$ can be removed.

In the following subsection, we introduce reduced *FA*'s as an encoding of *LBFA*'s.

4.2.1 Reduced *FA*'s

Definition 4.24 (*RFA*): A reduced *FA* (*RFA*) is a 7-tuple $(Q, V, follow, first, last, null, Qmap)$ where

- Q is a finite set of states,
- V is a alphabet,
- $follow \in \mathcal{P}(Q \times Q)$ is a follow relation (replacing the transition relation),
- $first \subseteq Q$ is a set of initial states (replacing $T(s)$ in an *LBFA*),
- $last \subseteq Q$ is a set of final states,
- $null \in \{true, false\}$ is a Boolean value (encoding $s \in F$ in an *LBFA*), and
- $Qmap \in \mathcal{P}(Q \times V)$ maps each state to exactly one symbol (it is also viewed as $Qmap \in Q \longrightarrow V$, and its inverse as $Qmap^{-1} \in V \not\longrightarrow \mathcal{P}(Q)$).

□

Definition 4.25 (Isomorphism of *RFA*'s): We extend isomorphism ($\cong$) to *RFA*'s. □

Definition 4.26 (Reversal of *RFA*'s): Reversal of *RFA*'s is given by postfix (superscript) function $R \in RFA \longrightarrow RFA$ defined as:

$$(Q, V, follow, first, last, null, Qmap)^R = (Q, V, follow^R, last, first, null, Qmap)$$

□

Definition 4.27 (Extending reversal to $[RFA]_{\cong}$): We extend reversal to $[RFA]_{\cong} \longrightarrow [RFA]_{\cong}$ as $([P]_{\cong})^R = [P^R]_{\cong}$. □

We can now give isomorphisms between $[LBFA]_{\cong}$ and $[RFA]_{\cong}$. These isomorphisms will be used to present a Σ-algebra of *RFA*'s.

Definition 4.28 (An isomorphism between $[LBFA]_{\cong}$ and $[RFA]_{\cong}$): We define isomorphism $encode \in [LBFA]_{\cong} \longrightarrow [RFA]_{\cong}$

$$
\begin{array}{lcll}
encode([(Q,V,T,\emptyset,\{s\},F)]_{\cong}) & = & \textbf{let} & Q' = Q \setminus \{s\} \\
& & \textbf{in} & \\
& & & [(Q',V,\bar{\pi}_2(T)\cap(Q'\times Q'),\pi_2(T(s)), \\
& & & \quad F\cap Q', s\in F,(\bar{\pi}_1(T))^R)]_{\cong} \\
& & \textbf{end} &
\end{array}
$$

and its inverse $decode \in [RFA]_{\cong} \longrightarrow [LBFA]_{\cong}$ as

$$
\begin{array}{lcll}
decode([R]_{\cong}) & = & \textbf{let} & (Q,V,follow,first,last,null,Qmap) = R \\
& & & s \text{ be a new state} \\
& & \textbf{in} & \\
& & & \begin{array}{ll}
\textbf{let} & T = \{(q_0,Qmap(q_1),q_1) : (q_0,q_1)\in follow\} \\
 & T' = \{(s,Qmap(q),q) : q \in first\} \\
 & F = last \cup \textbf{if } (null) \textbf{ then } \{s\} \textbf{ else } \emptyset \textbf{ fi} \\
\textbf{in} & \\
 & [(Q\cup\{s\},V,T\cup T',\emptyset,\{s\},F)]_{\cong} \\
\textbf{end} &
\end{array} \\
& & \textbf{end} &
\end{array}
$$

It is easy to verify that both of these functions are isomorphisms, and that *decode* is the inverse of *encode*. □

Given function *encode* and *decode*, we would like to obtain a Σ-algebra with $[RFA]_{\cong}$ as carrier (and a corresponding unique homomorphism $rfa \in RE \longrightarrow [RFA]_{\cong}$) such that the following diagram commutes:

$$
\begin{tikzcd}
RE \arrow[r, "lbfa"] \arrow[d, "rfa"'] & {[LBFA]_{\cong}} \\
{[RFA]_{\cong}} \arrow[ur, "decode"'] &
\end{tikzcd}
$$

We can now define a Σ-algebra of *RFA*'s; it will be cheaper to compute the *RFA* image of a regular expression and map the *RFA* to an *LBFA*, than to compute the *LBFA* directly. The operators of the Σ-algebra of *RFA*'s are defined using the *LBFA* operators and the isomorphisms *encode* and *decode*.

Definition 4.29 (Σ-algebra of *RFA*'s): The carrier is $[RFA]_{\cong}$. Given the operator requirement in the Σ-algebra of *LBFA*'s, the operator requirement in this Σ-algebra is:

- For binary operators, the argument representatives must have disjoint state sets.

The operators of the Σ-algebra of *RFA*'s are defined in terms of the operators of *LBFA*'s:

$$
\begin{array}{lcll}
C_{\epsilon,RFA} & = & encode(C_{\epsilon,LBFA}) & \\
C_{\emptyset,RFA} & = & encode(C_{\emptyset,LBFA}) & \\
C_{a,RFA} & = & encode(C_{a,LBFA}) & \text{(for all } a \in V) \\
C_{\cdot,RFA}([R_0]_{\cong},[R_1]_{\cong}) & = & encode \circ C_{\cdot,LBFA}(decode([R_0]_{\cong}), decode([R_1]_{\cong})) & \\
C_{\cup,RFA}([R_0]_{\cong},[R_1]_{\cong}) & = & encode \circ C_{\cup,LBFA}(decode([R_0]_{\cong}), decode([R_1]_{\cong})) & \\
C_{*,RFA}([R]_{\cong}) & = & encode \circ C_{*,LBFA}(decode([R]_{\cong})) & \\
C_{+,RFA}([R]_{\cong}) & = & encode \circ C_{+,LBFA}(decode([R]_{\cong})) & \\
C_{?,RFA}([R]_{\cong}) & = & encode \circ C_{?,LBFA}(decode([R]_{\cong})) &
\end{array}
$$

In full:

$$C_{\epsilon,RFA} = [(\emptyset, V, \emptyset, \emptyset, \emptyset, \mathit{true}, \emptyset)]_{\cong}$$

$$C_{\emptyset,RFA} = [(\emptyset, V, \emptyset, \emptyset, \emptyset, \mathit{false}, \emptyset)]_{\cong}$$

For all $a \in V$:

$$\begin{array}{lll} C_{a,RFA} & = & \mathbf{let} \quad q_0 \text{ be a new state} \\ & & \mathbf{in} \\ & & \quad [(\{q_0\}, V, \emptyset, \{q_0\}, \{q_0\}, \mathit{false}, \{(q_0, a)\})]_{\cong} \\ & & \mathbf{end} \end{array}$$

$$\begin{array}{lll} C_{\cdot,RFA}([R_0]_{\cong}, [R_1]_{\cong}) & = & \mathbf{let} \quad (Q_0, V, \mathit{follow}_0, \mathit{first}_0, \mathit{last}_0, \mathit{null}_0, \mathit{Qmap}_0) = R_0 \\ & & \qquad\ (Q_1, V, \mathit{follow}_1, \mathit{first}_1, \mathit{last}_1, \mathit{null}_1, \mathit{Qmap}_1) = R_1 \\ & & \mathbf{in} \\ & & \quad \mathbf{let} \quad \mathit{first}' = \mathit{first}_0 \cup \mathbf{if}\ (\mathit{null}_0)\ \mathbf{then}\ \mathit{first}_1\ \mathbf{else}\ \emptyset\ \mathbf{fi} \\ & & \qquad\qquad \mathit{last}' = \mathit{last}_1 \cup \mathbf{if}\ (\mathit{null}_1)\ \mathbf{then}\ \mathit{last}_0\ \mathbf{else}\ \emptyset\ \mathbf{fi} \\ & & \quad \mathbf{in} \\ & & \qquad [(Q_0 \cup Q_1, V, \mathit{follow}_0 \cup \mathit{follow}_1 \cup (\mathit{last}_0 \times \mathit{first}_1), \\ & & \qquad\quad \mathit{first}', \mathit{last}', \mathit{null}_0 \wedge \mathit{null}_1, \mathit{Qmap}_0 \cup \mathit{Qmap}_1)]_{\cong} \\ & & \quad \mathbf{end} \\ & & \mathbf{end} \end{array}$$

$$\begin{array}{lll} C_{\cup,RFA}([R_0]_{\cong}, [R_1]_{\cong}) & = & \mathbf{let} \quad (Q_0, V, \mathit{follow}_0, \mathit{first}_0, \mathit{last}_0, \mathit{null}_0, \mathit{Qmap}_0) = R_0 \\ & & \qquad\ (Q_1, V, \mathit{follow}_1, \mathit{first}_1, \mathit{last}_1, \mathit{null}_1, \mathit{Qmap}_1) = R_1 \\ & & \mathbf{in} \\ & & \quad [(Q_0 \cup Q_1, V, \mathit{follow}_0 \cup \mathit{follow}_1, \mathit{first}_0 \cup \mathit{first}_1, \\ & & \qquad \mathit{last}_0 \cup \mathit{last}_1, \mathit{null}_0 \vee \mathit{null}_1, \mathit{Qmap}_0 \cup \mathit{Qmap}_1)]_{\cong} \\ & & \mathbf{end} \end{array}$$

$$\begin{array}{lll} C_{*,RFA}([R]_{\cong}) & = & \mathbf{let} \quad (Q, V, \mathit{follow}, \mathit{first}, \mathit{last}, \mathit{null}, \mathit{Qmap}) = R \\ & & \mathbf{in} \\ & & \quad [(Q, V, \mathit{follow} \cup (\mathit{last} \times \mathit{first}), \mathit{first}, \mathit{last}, \mathit{true}, \mathit{Qmap})]_{\cong} \\ & & \mathbf{end} \end{array}$$

$$\begin{array}{lll} C_{+,RFA}([R]_{\cong}) & = & \mathbf{let} \quad (Q, V, \mathit{follow}, \mathit{first}, \mathit{last}, \mathit{null}, \mathit{Qmap}) = R \\ & & \mathbf{in} \\ & & \quad [(Q, V, \mathit{follow} \cup (\mathit{last} \times \mathit{first}), \mathit{first}, \mathit{last}, \mathit{null}, \mathit{Qmap})]_{\cong} \\ & & \mathbf{end} \end{array}$$

$$\begin{array}{lll} C_{?,RFA}([R]_{\cong}) & = & \mathbf{let} \quad (Q, V, \mathit{follow}, \mathit{first}, \mathit{last}, \mathit{null}, \mathit{Qmap}) = R \\ & & \mathbf{in} \\ & & \quad [(Q, V, \mathit{follow}, \mathit{first}, \mathit{last}, \mathit{true}, \mathit{Qmap})]_{\cong} \\ & & \mathbf{end} \end{array}$$

An $R \in \mathit{RFA}$ (the image of some $E \in \mathit{RE}$) in this Σ-algebra has the following interesting property:

- The number of states in R equals the number of (not necessarily distinct) symbols (of V) occuring in E. This follows from the fact that the operators $C_{a,RFA}$ (for all $a \in V$) are the only *RFA* operators that introduce new states. This property will be used in Section 4.5 to derive a practical implementation of the *RFA* operators.

We can also note the following about the operators:

- The operators do not depend on the choice of representatives of the equivalence classes.

- An important fact is that the operators of this Σ-algebra are symmetrical. That is, each operator is its own dual.

□

Definition 4.30 (Homomorphism from RE **to** $[RFA]_{\cong}$**):** We define $rfa \in RE \longrightarrow [RFA]_{\cong}$ to be the unique homomorphism from RE's to $[RFA]_{\cong}$. □

Property 4.31 (Homomorphism rfa**):** Since the operators of the Σ-algebra of RFA's are symmetrical (symmetrical functions are defined in Definition A.22), so is rfa. That is, $rfa \circ R(E) = R \circ rfa(E)$. □

In Section 4.5 practical implementations of the Σ-algebra of RFA's (in particular, of homomorphism rfa) are discussed.

4.2.2 The Berry-Sethi construction

Given the Σ-algebra of RFA's, we have the desired property that (for $E \in RE$):

$$lbfa(E) = decode \circ rfa(E)$$

We now present Berry and Sethi's FA construction.

Construction 4.32 (Berry-Sethi): Construction $BS \in RE \longrightarrow [FA]_{\cong}$ is defined as:

$$BS(E) = decode \circ rfa(E)$$

An automaton constructed using this function has the same properties as one constructed with function $lbfa$, namely:

- It is ϵ*-free*.
- It has a single start state.
- The single start state has no in-transitions.
- All in-transitions to a state are on the same symbol (of V).

In practice, function BS is cheaper to compute than $lbfa$. □

Remark 4.33: The history of this algorithm is somewhat complicated. The following account is given by Brüggemann-Klein [B-K93b]. Glushkov and McNaughton and Yamada simultaneously (and independently) discovered the same DFA construction [Glus61, MY60]. These papers use the same underlying ϵ*-free FA* construction to which they apply the subset construction[8]. Unfortunately, neither of them present the ϵ*-free FA* construction (without the subset construction) explicitly. The underlying ϵ*-free FA* construction was presented in some depth (with correctness arguments) by Berry and Sethi in [BS86, Alg. 4.4]. In their paper, Berry and Sethi also relate the construction to the Brzozowski construction (Brzozowski's construction appears as Construction 5.34 in this paper).

In this paper, we adopt the convention that the ϵ*-free FA* construction (without subset construction) is named after Berry and Sethi, while the construction with the subset construction is named after McNaughton, Yamada, and Glushkov. □

Example 4.34 (Berry-Sethi): A representative of the equivalence class $BS((a \cup \epsilon)b^*)$ is shown in Figure 4 appearing on page 32. This is the same FA as in Example 4.23. (This follows from the fact that the Berry-Sethi construction and the $LBFA$ Σ-algebra are commuting ways of arriving at the same FA isomorphism class). □

[8] The underlying construction may actually produce a nondeterministic finite automata.

It is possible to find a composition of functions that commutes with *lbfa* (and therefore $decode \circ rfa$) and is cheaper to compute in practice. We first give some required definitions.

Definition 4.35 (Non-isomorphic mapping from $[RFA]_{\cong}$ to $[FA]_{\cong}$): Function $convert \in [RFA]_{\cong} \longrightarrow [FA]_{\cong}$ is defined as:

$$\begin{array}{lcl} convert([R]_{\cong}) & = & \textbf{let} \quad (Q, V, follow, first, last, null, Qmap) = R \\ & & \textbf{in} \\ & & \qquad \textbf{let} \quad T = \{(q_0, Qmap(q_1), q_1) : (q_0, q_1) \in follow\} \\ & & \qquad \textbf{in} \quad\ [(Q, V, T, \emptyset, first, last)]_{\cong} \\ & & \qquad \textbf{end} \\ & & \textbf{end} \end{array}$$

An important property of this function is that:

$$(\forall\ E : E \in RE : \mathcal{L}_{FA} \circ convert \circ rfa(E) = V^{-1}\mathcal{L}_{RE}(E))$$

This follows from the fact that *convert* simply discards the transitions that would be out of the start state. Function *convert* does not add any new states, unlike function *decode* which adds a new start state. □

Definition 4.36 (Adding a begin-marker): Define function $marker_b \in RE \longrightarrow RE$ as:

$$marker_b(E) = \$ \cdot E$$

Where \$ is an alphabet symbol, called a begin-marker. (In the literature, it is usually assumed — for no particular reason — that symbol \$ does not occur in regular expression E.) This function satisfies the obvious property that:

$$(\forall\ E : E \in RE : \mathcal{L}_{RE}(marker_b(E)) = \{\$\}\mathcal{L}_{RE}(E))$$

□

Given functions $marker_b$, *rfa*, *convert*, and the following important property, we can construct an efficient alternative to homomorphism *lbfa*.

Property 4.37 (Functions $marker_b$, *rfa*, and *convert*): Because of the properties of *convert* and $marker_b$, we can show that:

$$\begin{array}{ll} & convert \circ rfa \circ marker_b(E) \\ = & \quad \{\text{Definition of } marker_b\} \\ & convert \circ rfa(\$ \cdot E) \\ = & \quad \{\text{Definitions of } rfa,\ C_{\cdot,RFA}\} \\ & convert \circ C_{\cdot,RFA}(rfa(\$), rfa(E)) \\ = & \quad \{\text{Definitions of } rfa,\ C_{\$,RFA}\} \\ & convert \circ C_{\cdot,RFA}(C_{\$,RFA}, rfa(E)) \\ = & \quad \{\text{Definitions of } convert,\ C_{\cdot,RFA},\ C_{\$,RFA},\ rfa, \text{ and } decode\} \\ & decode \circ rfa(E) \\ = & \quad \{\text{Commutativity}\} \\ & lbfa(E) \end{array}$$

The composite $convert \circ rfa \circ marker_b$ is an alternative (and in practice, cheaper) implementation of *lbfa*. □

The fact that $convert \circ rfa \circ marker_b$ is a construction is depicted in the following commuting diagram:

$$\begin{tikzcd} RE \arrow[r, "lbfa"] \arrow[d, "marker_b"'] & {[LBFA]_{\cong}} \\ RE \arrow[r, "rfa"] & {[RFA]_{\cong}} \arrow[u, "convert"'] \end{tikzcd}$$

Construction 4.38 (A variation on the Berry-Sethi construction): Instead of constructing an *FA* using the functions *lbfa* or *BS*, it is cheaper in practice to use the composite function

$$convert \circ rfa \circ marker_b(E)$$

□

4.2.3 The McNaughton-Yamada-Glushkov construction

Since the Berry-Sethi construction produces an *ϵ-free* (possibly nondeterministic) *FA*, we now consider making the resulting *FA* deterministic.

Construction 4.39 (McNaughton-Yamada-Glushkov): (We assume that the composite function $useful_s \circ subset$ is extended to $[FA]_{\cong} \longrightarrow [DFA]_{\cong}$.) The McNaughton-Yamada-Glushkov *DFA* construction is $MYG \in RE \longrightarrow [DFA]_{\cong}$, defined as:

$$MYG(E) = useful_s \circ subset \circ decode \circ rfa(E)$$

A *DFA* produced by *MYG* is *Complete* (by a property of $useful_s \circ subset$). A practical implementation is given in Algorithm 4.42 (given below), which implements $useful_s \circ subset \circ decode$. Homomorphism *rfa* can be implemented using the techniques described in Section 4.5. This algorithm is the same[9] as that given by McNaughton and Yamada [MY60, Construction method on pg. 44]. □

Example 4.40 (McNaughton-Yamada-Glushkov): In the case of $(a \cup \epsilon)b^* \in RE$, the Berry-Sethi construction produces a deterministic *FA*. Function *MYG* produces a similar *DFA*, with a sink state added to make it *Complete*. The state graph of a representative *DFA* of isomorphism class $MYG((a \cup \epsilon)b^*)$ is shown in Figure 5. □

Remark 4.41: The variation on the Berry-Sethi construction (Construction 4.38) can be used for a practical implementation of the McNaughton-Yamada-Glushkov construction. This would yield a construction not appearing in the literature. □

[9] The only difference is that the unrolled first iteration step is not presented explicitly in McNaughton and Yamada's paragraph describing their algorithm.

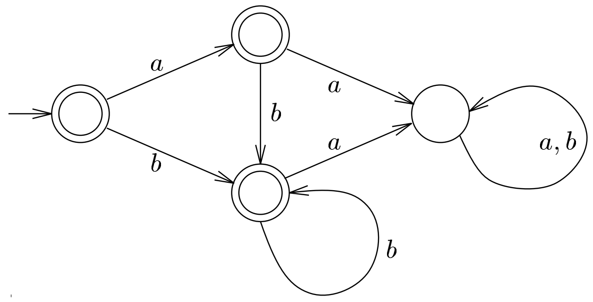


Figure 5: A representative *DFA* of the isomorphism class $MYG((a \cup \epsilon)b^*)$.

Composite function $useful_s \circ subset \circ decode$ can be implemented using Algorithm 2.45 (which implements $useful_s \circ subset$). Here, the first iteration is unrolled to accommodate the definition of function *decode*, and some obvious improvements have not yet been made).

$\{(Q, V, follow, first, last, null, Qmap) \in RFA\}$
let $S = \{\{s\}\} : s$ is a new state, $s \notin Q$;
$T := \emptyset$;
$D, U := \emptyset, S$;
let $u : u \in U$;
$D, U := D \cup \{u\}, U \setminus \{u\}$;
for $a : a \in V$ **do**
    $d := (\cup\ p : p \in u : \{q : q \in first \wedge Qmap(q) = a\})$;
    if $d \notin D \longrightarrow U := U \cup \{d\}$
    $[\!]\ d \in D \longrightarrow$ **skip**
    fi;
    $T := T \cup \{(u, a, d)\}$
rof;
do $U \neq \emptyset \longrightarrow$
    let $u : u \in U$;
    $D, U := D \cup \{u\}, U \setminus \{u\}$;
    for $a : a \in V$ **do**
        $d := (\cup\ p : p \in u : \{q : (p, q) \in follow \wedge Qmap(q) = a\})$;
        if $d \notin D \longrightarrow U := U \cup \{d\}$
        $[\!]\ d \in D \longrightarrow$ **skip**
        fi;
        $T := T \cup \{(u, a, d)\}$
    rof
od;
$F := \{d : d \in D \wedge d \cap last \neq \emptyset\} \cup$ **if** $(null)$ **then** S **else** $\emptyset$ **fi**
$\{[(D, V, T, \emptyset, S, F)]_{\cong} = useful_s \circ subset \circ$
    $decode([(Q, V, follow, first, last, null, Qmap)]_{\cong})\}$
$\{Complete(D, V, T, \emptyset, S, F)\}$

Some simplification gives the algorithm:

Algorithm 4.42 (McNaughton-Yamada-Glushkov):

$\{(Q, V, \mathit{follow}, \mathit{first}, \mathit{last}, \mathit{null}, \mathit{Qmap}) \in \mathit{RFA}\}$
let $S = \{\{s\}\} : s$ is a new state, $s \notin Q$;
$T := \emptyset$;
$D, U := S, \emptyset$;
for $a : a \in V$ **do**
  $d := \{q : q \in \mathit{first} \wedge \mathit{Qmap}(q) = a\}$
  $U := U \cup \{d\}$;
  $T := T \cup \{(\{s\}, a, d)\}$
rof;
do $U \neq \emptyset \longrightarrow$
  let $u : u \in U$;
  $D, U := D \cup \{u\}, U \setminus \{u\}$;
  for $a : a \in V$ **do**
    $d := (\cup\ p : p \in u : \{q : (p, q) \in \mathit{follow} \wedge \mathit{Qmap}(q) = a\})$;
    if $d \notin D \longrightarrow U := U \cup \{d\}$
    $\|\ d \in D \longrightarrow$ **skip**
    fi;
    $T := T \cup \{(u, a, d)\}$
  rof
od;
$F := \{d : d \in D \wedge d \cap \mathit{last} \neq \emptyset\} \cup$ **if** (null) **then** S **else** $\emptyset$ **fi**
$\{[(D, V, T, \emptyset, S, F)]_{\cong} = \mathit{useful}_s \circ \mathit{subset} \circ$
  $\mathit{decode}([(Q, V, \mathit{follow}, \mathit{first}, \mathit{last}, \mathit{null}, \mathit{Qmap})]_{\cong})\}$
$\{\mathit{Complete}(D, V, T, \emptyset, S, F)\}$

This algorithm is used in the McNaughton-Yamada construction [MY60].

4.3 The dual of the Berry-Sethi construction

The following commuting diagram gives a property of regular expressions and regular languages that will prove to be useful:

$$\begin{tikzcd} RE \arrow[r, "\mathcal{L}_{RE}"] \arrow[d, "R"'] & \mathcal{L}_{\text{reg}} \\ RE \arrow[r, "\mathcal{L}_{RE}"] & \mathcal{L}_{\text{reg}} \arrow[u, "R"'] \end{tikzcd}$$

In this diagram, the two reversal operators are different: one is reversal of *RE*'s, while the other is reversal of languages.

Given the definition of an *FA* construction f and the above diagram, we have the property that the dual of a construction is again a construction. That is, $R \circ f \circ R$ is also a construction. Such a dual construction is less efficient than f (since it involves two reversal functions), and we explore ways to efficiently implement the dual constructions.

Construction 4.43 (Right-biased): We can use $R \circ \mathit{lbfa} \circ R$ as a construction. For any given $E \in RE$, a representative of $R \circ \mathit{lbfa} \circ R(E)$ has the following properties (the properties are based upon those of the left-biased Σ-algebra):

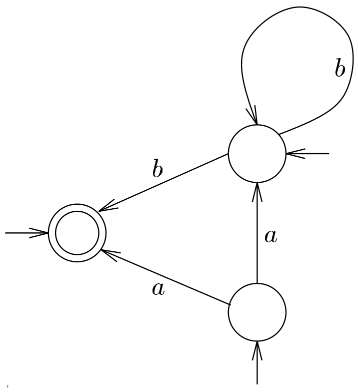


Figure 6: A representative *FA* of the isomorphism class $R \circ \mathit{lbfa} \circ R((a \cup \epsilon)b^*)$.

- It is ϵ-*free*.
- It has a single final state.
- The single final state has no out-transitions.
- All out-transitions from a state are on the same symbol (in V).

□

Example 4.44 (Right-biased construction): We construct a representative *FA* of the isomorphism class $R \circ \mathit{lbfa} \circ R((a \cup \epsilon)b^*)$ The representative is shown in Figure 6. □

To consider the dual of the Berry-Sethi construction, we combine the commuting diagrams of duals of a construction (above) and construction $\mathit{decode} \circ \mathit{rfa}$, giving:

$$\begin{tikzcd}
RE \arrow[d, "R"'] & & {[FA]_{\cong}} \arrow[r, "\mathcal{L}_{FA}"] & \mathcal{L}_{\mathrm{reg}} \\
RE \arrow[r, "\mathit{rfa}"] & {[RFA]_{\cong}} \arrow[r, "\mathit{decode}"] & {[FA]_{\cong}} \arrow[r, "\mathcal{L}_{FA}"] \arrow[u, "R"] & \mathcal{L}_{\mathrm{reg}} \arrow[u, "R"]
\end{tikzcd}$$

The source is the upper-left vertex, and the sink is the upper *FA* vertex.

The construction $R \circ \mathit{decode} \circ \mathit{rfa} \circ R$ (in this diagram) is still inefficient, requiring two redundant reversal operations. We can make it more efficient, by finding functions that form new paths in the commuting diagram.

From the definitions of the Σ-algebra of *RFA*'s (Definition 4.29) and homomorphism *rfa* (Definition 4.30) we know that the *RFA* operators are symmetrical, and so is *rfa*. In other words $\mathit{rfa} \circ R(E) = R \circ \mathit{rfa}(E)$. This allows us to add two new edges to the above commuting diagram; the resulting diagram is:

$$
\begin{tikzcd}
RE \arrow[r, "rfa"] \arrow[d, "R"'] & {[RFA]_{\cong}} \arrow[d, "R"'] & & {[FA]_{\cong}} \arrow[r, "\mathcal{L}_{FA}"] & \mathcal{L}_{\mathrm{reg}} \\
RE \arrow[r, "rfa"] & {[RFA]_{\cong}} \arrow[r, "decode"] & & {[FA]_{\cong}} \arrow[u, "R"] \arrow[r, "\mathcal{L}_{FA}"] & \mathcal{L}_{\mathrm{reg}} \arrow[u, "R"]
\end{tikzcd}
$$

Construction 4.45 (The dual of Berry-Sethi): The construction is:

$R \circ \mathit{decode} \circ R \circ \mathit{rfa}(E)$

This construction is the dual of the Berry-Sethi construction (Construction 4.32). □

We give $R \circ \mathit{decode} \circ R$ in full:

$$
\begin{array}{lcll}
R \circ \mathit{decode} \circ R([R]_{\cong}) & = & \textbf{let} & (Q, V, \mathit{follow}, \mathit{first}, \mathit{last}, \mathit{null}, \mathit{Qmap}) = R \\
& & & f \text{ be a new state} \\
& & \textbf{in} & \\
& & & \begin{array}{ll}
\textbf{let} & T = \{(q_0, \mathit{Qmap}(q_0), q_1) : (q_0, q_1) \in \mathit{follow}\} \\
& T' = \{(q, \mathit{Qmap}(q), f) : q \in \mathit{last}\} \\
& S = \mathit{first} \cup \textbf{if } (\mathit{null}) \textbf{ then } \{f\} \textbf{ else } \emptyset \textbf{ fi} \\
\textbf{in} & \\
& [(Q \cup \{f\}, V, T \cup T', \emptyset, S, \{f\})]_{\cong} \\
\textbf{end} &
\end{array} \\
& & \textbf{end} &
\end{array}
$$

The *FA* resulting from this construction is the same as given in Example 4.44

We can also consider improving the dual of the variation on the Berry-Sethi construction (Construction 4.38). We combine the commuting diagram showing the dual of a construction, with construction $\mathit{convert} \circ \mathit{rfa} \circ \mathit{marker}_b$, giving:

$$
\begin{tikzcd}
RE \arrow[d, "R"'] & & & {[FA]_{\cong}} \arrow[r, "\mathcal{L}_{FA}"] & \mathcal{L}_{\mathrm{reg}} \\
RE \arrow[r, "\mathit{marker}_b"] & RE \arrow[r, "rfa"] & {[RFA]_{\cong}} \arrow[r, "convert"] & {[FA]_{\cong}} \arrow[u, "R"] \arrow[r, "\mathcal{L}_{FA}"] & \mathcal{L}_{\mathrm{reg}} \arrow[u, "R"]
\end{tikzcd}
$$

Again, the source is the upper-left *RE* vertex, while the sink is the upper-right *FA* vertex.

Consider the composite function $R \circ \mathit{convert} \circ \mathit{rfa} \circ \mathit{marker}_b \circ R$. We begin with the two rightmost functions:

$$
\begin{array}{ll}
& \mathit{marker}_b \circ R(E) \\
= & \quad \{\text{Writing } R \text{ as postfix and superscript}\} \\
& \mathit{marker}_b(E^R) \\
= & \quad \{\text{Definition of } \mathit{marker}_b \text{ (Definition 4.36)}\} \\
& \$ \cdot (E^R) \\
= & \quad \{\text{Function } R \circ R \text{ is the identity (see Definition A.19)}\} \\
& R \circ R(\$ \cdot (E^R)) \\
= & \quad \{\text{Definition of } R \text{ on } \cdot \text{ regular expressions}\}
\end{array}
$$

$$
\begin{array}{ll}
 & R((E^R)^R \cdot (\$)^R) \\
= & \quad \{\text{Definition of } R \text{ and } R \circ R\} \\
 & R(E \cdot \$)
\end{array}
$$

To make this definition more concise, we define an end-marker function.

Definition 4.46 (Adding an end-marker): Define function $marker_e \in RE \longrightarrow RE$ as:

$$marker_e(E) = E \cdot \$$$

where \$ is assume to be a symbol in the alphabet. □

Property 4.47 ($marker_e$): Function $marker_e$ is the dual of function $marker_b$ (Definition 4.36):

$$marker_b \circ R(E) = R \circ marker_e(E)$$

□

With the above property, we can transform the above commuting diagram, by adding two new edges:

$$
\begin{tikzcd}
RE \arrow[r, "marker_e"] \arrow[d, "R"'] & RE \arrow[d, "R"'] & & {[FA]_{\cong}} \arrow[r, "\mathcal{L}_{FA}"] & \mathcal{L}\text{reg} \\
RE \arrow[r, "marker_b"] & RE \arrow[r, "rfa"] & {[RFA]_{\cong}} \arrow[r, "convert"] & {[FA]_{\cong}} \arrow[r, "\mathcal{L}_{FA}"] \arrow[u, "R"] & \mathcal{L}\text{reg} \arrow[u, "R"]
\end{tikzcd}
$$

The composite $R \circ convert \circ rfa \circ R \circ marker_e$ is no more efficient even with the use of $marker_e$. Fortunately, since rfa is symmetrical, we can replace $rfa \circ R$ by $R \circ rfa$, giving:

$$
\begin{tikzcd}
RE \arrow[r, "marker_e"] \arrow[d, "R"'] & RE \arrow[r, "rfa"] \arrow[d, "R"'] & {[RFA]_{\cong}} \arrow[d, "R"'] & {[FA]_{\cong}} \arrow[r, "\mathcal{L}_{FA}"] & \mathcal{L}\text{reg} \\
RE \arrow[r, "marker_b"] & RE \arrow[r, "rfa"] & {[RFA]_{\cong}} \arrow[r, "convert"] & {[FA]_{\cong}} \arrow[r, "\mathcal{L}_{FA}"] \arrow[u, "R"] & \mathcal{L}\text{reg} \arrow[u, "R"]
\end{tikzcd}
$$

The composite function $R \circ convert \circ R$ is particularly easy to present, using the definitions of R and $convert$ (Definition 4.35):

$$
\begin{array}{lll}
R \circ convert \circ R([R]_{\cong}) & = & \textbf{let} \quad (Q, V, follow, first, last, null, Qmap) = R \\
 & & \textbf{in} \\
 & & \qquad \begin{array}{ll} \textbf{let} & T = \{(q_0, Qmap(q_0), q_1) : (q_0, q_1) \in follow\} \\ \textbf{in} & [(Q, V, T, \emptyset, first, last)]_{\cong} \\ \textbf{end} & \end{array} \\
 & & \textbf{end}
\end{array}
$$

This leads to the following construction.

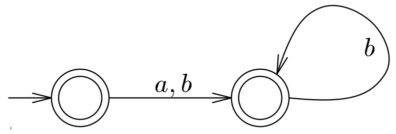


Figure 7: A representative *DFA* of the isomorphism class $ASU((a \cup \epsilon)b^*)$.

Construction 4.48 (The dual of the variation on the Berry-Sethi construction): The construction is:

$R \circ convert \circ R \circ rfa \circ marker_e(E)$

This construction is also presented very informally by Aho, Sethi, and Ullman [ASU86, Example 3.22, pg. 140]. There appears to be an error in item two of the three steps describing the construction in [ASU86]. Instead of

2. label each directed edge (i, j) by the symbol at position j, and

the step should read

2. label each directed edge (i, j) by the symbol at position i, and

□

Example 4.49 (The dual of the variation on the Berry-Sethi construction): A representative *FA* of the isomorphism class $R \circ convert \circ R \circ rfa \circ marker_e((a \cup \epsilon)b^*)$ is shown in Figure 6 on page 40. This is the same *FA* as in Example 4.44. □

4.3.1 The Aho-Sethi-Ullman *DFA* construction

In order to obtain a (possibly non-*Complete*) *DFA* we use the composite function $useful_s \circ subsetopt$ (given in Definition 2.44), extended to $[FA]_{\cong} \longrightarrow [DFA]_{\cong}$.

We can immediately give the Aho-Sethi-Ullman *DFA* construction using this composite function.

Construction 4.50 (Aho-Sethi-Ullman): The construction is $ASU \in RE \longrightarrow [DFA]_{\cong}$ defined as:

$ASU(E) = useful_s \circ subsetopt \circ R \circ convert \circ R \circ rfa \circ marker_e(E)$

Algorithm 4.52 (given below) is an imperative program implementing

$useful_s \circ subsetopt \circ R \circ convert \circ R$

Homomorphism *rfa* can be implemented using the techniques described in Section 4.5, and function $marker_e$ is trivial to implement. The Aho-Sethi-Ullman algorithm is given in [ASU86, Alg. 3.5, Fig. 3.44]. □

Example 4.51 (Aho-Sethi-Ullman): We give a representative *DFA* of the isomorphism class $ASU((a \cup \epsilon)b^*)$. The state graph is shown in Figure 7. □

We compose $useful_s \circ subsetopt$ (as implemented by Algorithm 2.46) with $R \circ convert \circ R$. The resulting algorithm is simplified in a similar way to the McNaughton-Yamada-Glushkov algorithm (Algorithm 4.42).

Algorithm 4.52 (Aho-Sethi-Ullman):

$\{(Q, V, follow, first, last, null, Qmap) \in RFA\}$
$S, T := (\textbf{if } (first \neq \emptyset) \textbf{ then } \{first\} \textbf{ else } \emptyset \textbf{ fi}), \emptyset;$
$D, U := \emptyset, S;$
do $U \neq \emptyset \longrightarrow$
  let $u : u \in U;$
  $D, U := D \cup \{u\}, U \setminus \{u\};$
  for $a : a \in V \wedge (\exists\ q : q \in u : Qmap(q) = a \wedge follow(q) \neq \emptyset)$ **do**
    $d := (\cup\ q : q \in u \wedge Qmap(q) = a : follow(q));$
    if $d \notin D \longrightarrow U := U \cup \{d\}$
    $\|$ $d \in D \longrightarrow$ **skip**
    fi;
    $T := T \cup \{(u, a, d)\}$
  rof
od;
$F := \{d : d \in D \wedge d \cap last \neq \emptyset\}$
$\{[(D, V, T, \emptyset, S, F)]_{\cong} = useful_s \circ subsetopt \circ$
  $R \circ convert \circ R([(Q, V, follow, first, last, null, Qmap)]_{\cong})\}$

4.4 Extending regular expressions

For some regular languages, the regular expressions denoting the language can be can be considerably more succinct when operators such as intersection ($\cap$) and complement ($\neg$) are available in *RE*'s. Without formally adding them to the signature Σ, we briefly consider how to implement operator $\cap$ in the left-biased Σ-algebra of *FA*'s.

Definition 4.53 (Extended regular expressions and their languages): The set of extended regular expressions (over alphabet V), *ERE*, and the languages they denote, are exactly as *RE*, with the addition of the operators $\cap \in ERE \times ERE \longrightarrow ERE$ (an infix operator) and $\neg \in ERE \longrightarrow ERE$ (a prefix operator). Operator $\cap$ has the same precedence as $\cup$, while $\neg$ has higher precedence than $*$. The language of an *ERE* is defined using the function $\mathcal{L}_{ERE} \in ERE \longrightarrow \mathcal{L}_{\mathrm{reg}}$ which is as function $\mathcal{L}_{RE}$, with the extensions

$$\begin{aligned} \mathcal{L}_{ERE}(E_0 \cap E_1) &= \mathcal{L}_{ERE}(E_0) \cap \mathcal{L}_{ERE}(E_1) \\ \mathcal{L}_{ERE}(\neg E_0) &= V^* \setminus \mathcal{L}_{ERE}(E_0) \end{aligned}$$

$\square$

Remark 4.54: The Σ-algebra definition of regular expressions are not used in this section as the algebraic structure is not needed. $\square$

Definition 4.55 (Intersection of *LBFA*'s): In defining intersection, we assume that the two arguments have been constructed in the Σ-algebra of *LBFA*. In particular, we require that for each state, all in-transitions are on the same symbol. Assuming the argument representatives have disjoint state sets, one possible implementation of the operator is[10]:

$$
\begin{array}{rcl}
C_{\cap,LBFA}([M_0]_{\cong},[M_1]_{\cong}) & = & \textbf{let} \quad (Q_0,V,T_0,\emptyset,\{s_0\},F_0)=M_0 \\
& & \qquad\;\; (Q_1,V,T_1,\emptyset,\{s_1\},F_1)=M_1 \\
& & \qquad\;\; q_0 \text{ be a new state} \\
& & \qquad\;\; N=\epsilon\in(\mathcal{L}_{FA}(M_0)\cap\mathcal{L}_{FA}(M_1)) \\
& & \textbf{in} \\
& & \quad \textbf{let} \quad Q'=\{q_0\}\cup(\cup\ b: b\in V:\pi_2(T_0(b))\times\pi_2(T_1(b))) \\
& & \qquad\qquad T'(a)=\{q_0\}\times(T_0(s_0,a)\times T_1(s_1,a)) \\
& & \qquad\qquad\quad \cup\{((p,q),(p',q')):(p,p')\in T_0(a)\wedge p\neq s_0 \\
& & \qquad\qquad\quad \wedge\,(q,q')\in T_1(a)\wedge q\neq s_1 \\
& & \qquad\qquad\quad \wedge\,(\exists\ b: b\in V: p\in\pi_2(T_0(b))\wedge q\in\pi_2(T_1(b)))\} \\
& & \quad \textbf{in} \\
& & \qquad\qquad [(Q',V,T',\emptyset,\{q_0\},(F_0\times F_1)\cap Q' \\
& & \qquad\qquad\quad \cup\ \textbf{if}\ (N)\ \textbf{then}\ \{q_0\}\ \textbf{else}\ \emptyset\ \textbf{fi})]_{\cong} \\
& & \quad \textbf{end} \\
& & \textbf{end}
\end{array}
$$

The expression

$$Q'=\{q_0\}\cup(\cup\ b: b\in V:\pi_2(T_0(b))\times\pi_2(T_1(b)))$$

in the **let** clause deserves some explanation. A state in the constructed *LBFA* is either the new state q_0, or a pair of states (p,q) where p and q ($p\neq s_0, q\neq s_1$) are from M_0 and M_1 respectively. If p and q do not have an in-transition on the same symbol, the state (p,q) will be start-unreachable in the constructed *LBFA*. For this reason, it is omitted. The definition of the transition relation is similar. The constructed *LBFA* is sometimes called the *cross-product LBFA*. Although the operator removes most start-unreachable states, some may still remain. □

We can now present an intersection operator for *RFA*'s.

Definition 4.56 (Intersection of *RFA*'s): We define intersection of *RFA*'s as:

$$C_{\cap,RFA}([R_0]_{\cong},[R_1]_{\cong})=encode\circ C_{\cap,LBFA}(decode([R_0]_{\cong}),decode([R_1]_{\cong}))$$

In full:

$$
\begin{array}{rcl}
C_{\cap,RFA}([R_0]_{\cong},[R_1]_{\cong}) & = & \textbf{let} \quad (Q_0,V,follow_0,first_0,last_0,null_0,Qmap_0)=R_0 \\
& & \qquad\;\; (Q_1,V,follow_1,first_1,last_1,null_1,Qmap_1)=R_1 \\
& & \textbf{in} \\
& & \quad \textbf{let} \quad Q'=(\cup\ b: b\in V: Qmap_0^{-1}(b)\times Qmap_1^{-1}(b)) \\
& & \qquad\qquad follow'=\{((p,q),(p',q')):(p,p')\in follow_0 \\
& & \qquad\qquad\quad \wedge\,(q,q')\in follow_1 \\
& & \qquad\qquad\quad \wedge\ Qmap_0(p)=Qmap_1(q) \\
& & \qquad\qquad\quad \wedge\ Qmap_0(p')=Qmap_1(q')\} \\
& & \qquad\qquad first'=\{(p,q):p\in first_0\wedge q\in first_1 \\
& & \qquad\qquad\quad \wedge\ Qmap_0(p)=Qmap_0(q)\} \\
& & \qquad\qquad Qmap'^{-1}(a)=Qmap_0^{-1}(a)\times Qmap_1^{-1}(a) \\
& & \quad \textbf{in} \\
& & \qquad\qquad [(Q',V,follow',first',(last_0\times last_1)\cap Q', \\
& & \qquad\qquad\quad null_0\wedge null_1, Qmap')]_{\cong} \\
& & \quad \textbf{end} \\
& & \textbf{end}
\end{array}
$$

Note that this operator is symmetrical. □

[10] The definition presented here is intentionally clumsy, making it easier to present intersection of *RFA*'s

In their original paper [MY60, Section IV], McNaughton and Yamada attempt to define intersection. Unfortunately, their informal presentation is difficult to understand. Subsequent presentations of the *RFA* operators have all omitted intersection. For example, Berry and Sethi note (in [BS86, Remark 3.1]):

> "The approach ... does not extend to regular expressions with intersection and complementation operators."

We can construct an *RFA* operator for any regular operator (an operator on languages that preserves the regularity property) for which we can construct an *LBFA* operator. Examples of such operators are intersection, symmetrical difference, complement, asymmetrical difference, and prefix closure.

4.5 Efficiently computing with *RFA*'s

In this section, we consider some practical methods for constructing *RFA*'s. The first subsection considers a practical implementation of the (Σ-algebra of) *RFA* operators, while the second subsection introduces some improvements (due to Chang, Paige, and Brüggemann-Klein) to the *RFA* operators.

4.5.1 A practical implementation of the *RFA* operators

In an *RFA*, the states are mapped to their corresponding symbol (of V) by the seventh component (usually called *Qmap*) of the *RFA*. This seventh component would be redundant if the states and symbols were in a one-to-one correspondence. Furthermore, the symbols could then be used as the states. In this subsection, we explore this encoding method, and the requirements on the *RE*'s for this method to work. We will also be defining a new, restricted, mapping $rfa' \in RE \not\longrightarrow RFA$. We will be able to use this mapping for regular expressions in which each alphabet symbol occurs no more than once.

We first define an important auxiliary function.

Definition 4.57 (Occurrences of symbols in *RE*'s): We define function $Occ \in RE \longrightarrow \mathcal{P}(V)$ such that $Occ(E)$ is the set of symbols (of V) occurring in E. We can also define Occ recursively as follows:

$$
\begin{array}{lcll}
Occ(\epsilon) & = & \emptyset & \\
Occ(\emptyset) & = & \emptyset & \\
Occ(a) & = & \{a\} & \text{(for } a \in V) \\
Occ(E \cdot F) & = & Occ(E) \cup Occ(F) & \\
Occ(E \cup F) & = & Occ(E) \cup Occ(F) & \\
Occ(E^*) & = & Occ(E) & \\
Occ(E^+) & = & Occ(E) & \\
Occ(E^?) & = & Occ(E) &
\end{array}
$$

□

Definition 4.58 (*RRE*): We define $RRE \subset RE$ as the smallest set satisfying:

- $\epsilon \in RRE$,
- $\emptyset \in RRE$,
- $a \in RRE$ (for $a \in V$),
- if $E, F \in RRE$, and $Occ(E) \cap Occ(F) = \emptyset$ then $E \cdot F \in RRE$ and $E \cup F \in RRE$, and
- if $E \in RRE$ then $E^* \in RRE$, $E^+ \in RRE$, and $E^? \in RRE$.

Intuitively, *RRE* (for restricted regular expressions) denotes the set of all $E \in RE$ such that each symbol (of V) occurs no more than once in E. □

Example 4.59 (*RRE*): A *RRE* is $(a \cup \epsilon)b^*$. □

In order to give our alternative *RE* to *RFA* mapping, $rfa' \in RRE \longrightarrow RFA$, we first define some more auxiliary functions. The definitions of these functions also follow directly from the *RFA* operators.

Definition 4.60 (*First*): We define $First \in RE \longrightarrow \mathcal{P}(V)$ recursively (recall from Example 3.20 that $Null(E) \equiv (\epsilon \in \mathcal{L}_{RE}(E))$):

$$\begin{array}{lcll}
First(\epsilon) & = & \emptyset & \\
First(\emptyset) & = & \emptyset & \\
First(a) & = & \{a\} & (\text{for } a \in V) \\
First(E \cdot F) & = & First(E) \cup \textbf{if } (Null(E)) \textbf{ then } First(F) \textbf{ else } \emptyset \textbf{ fi} & \\
First(E \cup F) & = & First(E) \cup First(F) & \\
First(E^*) & = & First(E) & \\
First(E^+) & = & First(E) & \\
First(E^?) & = & First(E) &
\end{array}$$

This definition follows directly from the *first* tuple element of the *RFA* operator definitions. □

Remark 4.61: It is useful to have an intuitive understanding of function *First*. $First(E)$ is the set of all symbols that can occur as the first symbol of a string in $\mathcal{L}_{RE}(E)$. □

Definition 4.62 (*Last*): Function *Last* is defined to be the dual of *First*. □

Remark 4.63: $Last(E)$ is the set of all symbols that can occur as the last symbol of a string in $\mathcal{L}_{RE}(E)$. □

Definition 4.64 (*Follow*): We define $Follow \in RE \longrightarrow \mathcal{P}(V \times V)$ recursively:

$$\begin{array}{lcll}
Follow(\epsilon) & = & \emptyset & \\
Follow(\emptyset) & = & \emptyset & \\
Follow(a) & = & \emptyset & (\text{for } a \in V) \\
Follow(E \cdot F) & = & Follow(E) \cup Follow(F) \cup (Last(E) \times First(F)) & \\
Follow(E \cup F) & = & Follow(E) \cup Follow(F) & \\
Follow(E^*) & = & Follow(E) \cup (Last(E) \times First(E)) & \\
Follow(E^+) & = & Follow(E) \cup (Last(E) \times First(E)) & \\
Follow(E^?) & = & Follow(E) &
\end{array}$$

This definition follows directly from the *follow* tuple element of the *RFA* operator definitions. □

Remark 4.65: For $a, b \in V$, $(a, b) \in Follow(E)$ is equivalent to ab being a substring of some string in $\mathcal{L}_{RE}(E)$. □

Example 4.66 (*First, Last, Null, Follow*): We use the regular expression $(a \cup \epsilon)b^*$ (from Example 3.15):

$$\begin{array}{lcl}
First((a \cup \epsilon)b^*) & = & \{a, b\} \\
Last((a \cup \epsilon)b^*) & = & \{a, b\} \\
Null((a \cup \epsilon)b^*) & = & true \\
Follow((a \cup \epsilon)b^*) & = & \{(a, b), (b, b)\}
\end{array}$$

□

We now have the auxiliary functions required for the definition of rfa'.

Definition 4.67 (Function $rfa' \in RRE \longrightarrow RFA$): The definition of rfa' is straightforward:

$$rfa'(E) = (Occ(E), V, Follow(E), First(E), Last(E), Null(E), I_V)$$

where I_V is the identity function on alphabet symbols. □

Example 4.68 (rfa'): Using the results of the above example, we have:

$$rfa'((a \cup \epsilon)b^*) = (\{a,b\},\{a,b\},\{(a,b),(b,b)\},\{a,b\},\{a,b\}, true, \{(a,a),(b,b)\})$$

□

Property 4.69 (rfa'): Given $E \in RRE$ then $rfa(E) = [rfa'(E)]_{\cong}$. □

Function rfa' is convenient, as all of the auxiliary functions can easily be computed bottom-up on the structure of E.

Construction 4.70 (An encoding of BS): The method of constructing an RFA (using rfa') leads to a particularly concise definition of BS. For example, we define $BSenc \in RRE \longrightarrow FA$:

$$\begin{array}{lll}
BSenc(E) & = & \textbf{let} \quad s \text{ be a new state} \\
 & & \textbf{in} \\
 & & \quad \begin{array}{ll}
 \textbf{let} & T = \{(a,b,b) : (a,b) \in Follow(E)\} \\
 & T' = \{(s,a,a) : a \in First(E)\} \\
 & F = Last(E) \cup \textbf{if } (Null(E)) \textbf{ then } \{s\} \textbf{ else } \emptyset \textbf{ fi} \\
 \textbf{in} & \\
 & (Occ(E) \cup \{s\}, V, T \cup T', \emptyset, \{s\}, F) \\
 \textbf{end} &
 \end{array} \\
 & & \textbf{end}
\end{array}$$

□

Remark 4.71: Compare the definition of $BSenc$ to the definition of $decode$ (Definition 4.28). □

Property 4.72 (Construction $BSenc$): For $E \in RRE$:

$$[BSenc(E)]_{\cong} = BS(E)$$

□

Remark 4.73: By inspection, we see that (for $E \in RRE$) the FA $BSenc(E)$ (equivalently $BS(E)$) is deterministic. This implies that:

$$MYG(E) = complete \circ BS(E)$$

□

Remark 4.74: In Section 5.4 we will show that Brzozowski's construction (with an appropriate encoding) produces a DFA (from an $E \in RRE$) that is isomorphic to the one produced by $BSenc$ (and therefore BS). □

Similarly, the Aho-Sethi-Ullman algorithm becomes quite concise (from Algorithm 4.52):

$\{E \in RRE\}$
$E' := marker_e(E);$
$S, T :=$ (**if** $(First(E') \neq \emptyset)$ **then** $\{First(E')\}$ **else** $\emptyset$ **fi**)$, \emptyset;$
$D, U := \emptyset, S;$
do $U \neq \emptyset \longrightarrow$
  let $u : u \in U;$
  $D, U := D \cup \{u\}, U \setminus \{u\};$
  for $a : a \in u \wedge Follow(E')(a) \neq \emptyset$ **do**
    $d := (Follow(E'))(a);$
    if $d \notin D \longrightarrow U := U \cup \{d\}$
    $[\!]\ d \in D \longrightarrow$ **skip**
    fi;
    $T := T \cup \{(u, a, d)\}$
  rof
od;
$F := \{d : d \in D \wedge \$ \in d\}$
$\{\mathcal{L}_{FA}(D, V, T, \emptyset, S, F) = \mathcal{L}_{RE}(E)\}$

This algorithm is very similar to the one given in [ASU86].

The only problem remaining is how to deal with an $E \in RE$ when $E \notin RRE$. The method usually used is to "mark" the symbols of E (perhaps with an integer subscript), making each symbol unique. For example, $(a^+ \cup ab) \notin RRE$ but after marking we get $(a_1^+ \cup a_2 b_3) \in RRE$. Once the corresponding *FA* is constructed from the marked regular expression, the marks are removed (the *FA* is "unmarked") and the *FA* accepts $\mathcal{L}_{RE}(E)$. There are a few different styles of marking. For example, consider $a^+ \cup ab$: McNaughton-Yamada mark this as $a_1^+ \cup a_2 b_1$, Berry-Sethi use $a_1^+ \cup a_2 b_3$, and Aho-Sethi-Ullman use $1^+ \cup 23$.

The only disadvantage to the use of marking to encode *RFA* computation is that marking is unable to deal with some of the other regular operators, such as intersection, and complementation. For all $E, F \in RRE$ we have the property that $\mathcal{L}_{RE}(E \cap F) = \mathcal{L}_{RE}(E) \cap \mathcal{L}_{RE}(F) = \emptyset$. For example, given[11] $aa \cap a^*$ (with language $\mathcal{L}_{RE}(aa \cap a^*) = \mathcal{L}_{RE}(aa) \cap \mathcal{L}_{RE}(a^*) = \{aa\}$). After marking we get $a_1 a_2 \cap a_3^*$ after marking (with $\mathcal{L}_{RE}(a_1 a_2 \cap a_3^*) = \emptyset$). In Section 4.4 we saw how these operators can be readily implemented with *RFA*'s (without the encoding scheme of this section).

The approach presented in this subsection is essentially due to McNaughton and Yamada [MY60], Glushkov [Glus61], and Berry and Sethi [BS86]. The presentations in [B-K93a, Section 2], [BS86], [tEvG93], and [ASU86, Fig. 3.40, pp. 134–141] are particularly clear. Those interested in a rigorous treatment of this approach to *RFA*'s can refer to the paper of ten Eikelder and van Geldrop [tEvG93].

4.5.2 More efficient *RFA* operators

The definition of the *RFA* operators may still result in inefficient implementation. In particular, Brüggemann-Klein and Chang and Paige found that the implementation of the $(\cup)$ in the *RFA* operators may require more than constant time [B-K93a, Chan92, CP92]. In most cases the arguments (of $\cup$) are disjoint; the only possible exception is the union *follow* $\cup$ (*last* $\times$ *first*), appearing in the $C_{*,RFA}$ and $C_{+,RFA}$ operators. Two solutions to this problem will be presented here.

Convention 4.75 (Constant time union): We use the symbol $\uplus$ to denote union where the arguments to $\uplus$ are assumed to be disjoint. $\square$

---

[11] Here we assume, for the moment, that $\mathcal{L}_{RE}$ can deal with the intersection operator.

The first solution was proposed by Chang and Paige [Chan92, CP92].

Definition 4.76 (Chang-Paige *RFA*): We add an eighth component W to each *RFA*

$$(Q, V, \mathit{follow}, \mathit{first}, \mathit{last}, \mathit{null}, \mathit{Qmap}, W)$$

such that

$$W = (\mathit{last} \times \mathit{first}) \setminus \mathit{follow}$$

These modified *RFA*'s will be called Chang-Paige *RFA*'s, and are denoted by *RFA*$'$. □

We only give the new operators, instead of the Σ-algebra. The operators follow directly from the above definition.

Construction 4.77 (Operators of the Σ-algebra of Chang-Paige *RFA*'s): As usual, the operator requirement is:

- For binary operators, the representatives have disjoint state sets.

$$\begin{array}{lcl} C_{\epsilon,RFA'} & = & [(\emptyset, V, \emptyset, \emptyset, \emptyset, \mathit{true}, \emptyset, \emptyset)]_{\cong} \\ C_{\emptyset,RFA'} & = & [(\emptyset, V, \emptyset, \emptyset, \emptyset, \mathit{false}, \emptyset, \emptyset)]_{\cong} \end{array}$$

$$\begin{array}{lcll} C_{a\in V,RFA'} & = & \textbf{let} & q_0 \text{ be a new state} \\ & & \textbf{in} & \\ & & & [(\{q_0\}, V, \emptyset, \{q_0\}, \{q_0\}, \mathit{false}, \{(q_0, a)\}, \{(q_0, q_0)\})]_{\cong} \\ & & \textbf{end} & \end{array}$$

$$\begin{array}{lcll} C_{\cdot,RFA'}([R_0]_{\cong}, [R_1]_{\cong}) & = & \textbf{let} & (Q_0, V, \mathit{follow}_0, \mathit{first}_0, \mathit{last}_0, \mathit{null}_0, \mathit{Qmap}_0, W_0) = R_0 \\ & & & (Q_1, V, \mathit{follow}_1, \mathit{first}_1, \mathit{last}_1, \mathit{null}_1, \mathit{Qmap}_1, W_1) = R_1 \\ & & \textbf{in} & \\ & & & \begin{array}{ll} \textbf{let} & \mathit{first}' = \mathit{first}_0 \uplus \textbf{if } (\mathit{null}_0) \textbf{ then } \mathit{first}_1 \textbf{ else } \emptyset \textbf{ fi} \\ & \mathit{last}' = \mathit{last}_1 \uplus \textbf{if } (\mathit{null}_1) \textbf{ then } \mathit{last}_0 \textbf{ else } \emptyset \textbf{ fi} \\ & W' = \textbf{if } (\mathit{null}_1) \textbf{ then } W_0 \textbf{ else } \emptyset \textbf{ fi} \\ & W'' = \textbf{if } (\mathit{null}_0) \textbf{ then } W_1 \textbf{ else } \emptyset \textbf{ fi} \\ \textbf{in} & \\ & [(Q_0 \uplus Q_1, V, \mathit{follow}_0 \uplus \mathit{follow}_1 \uplus (\mathit{last}_0 \times \mathit{first}_1), \\ & \quad \mathit{first}', \mathit{last}', \mathit{null}_0 \wedge \mathit{null}_1, \mathit{Qmap}_0 \uplus \mathit{Qmap}_1, \\ & \quad (\mathit{last}_1 \times \mathit{first}_0) \uplus W' \uplus W'')]_{\cong} \\ \textbf{end} & \end{array} \\ & & \textbf{end} & \end{array}$$

$$\begin{array}{lcll} C_{\cup,RFA'}([R_0]_{\cong}, [R_1]_{\cong}) & = & \textbf{let} & (Q_0, V, \mathit{follow}_0, \mathit{first}_0, \mathit{last}_0, \mathit{null}_0, \mathit{Qmap}_0, W_0) = R_0 \\ & & & (Q_1, V, \mathit{follow}_1, \mathit{first}_1, \mathit{last}_1, \mathit{null}_1, \mathit{Qmap}_1, W_1) = R_1 \\ & & \textbf{in} & \\ & & & [(Q_0 \uplus Q_1, V, \mathit{follow}_0 \uplus \mathit{follow}_1, \mathit{first}_0 \uplus \mathit{first}_1, \\ & & & \quad \mathit{last}_0 \uplus \mathit{last}_1, \mathit{null}_0 \vee \mathit{null}_1, \mathit{Qmap}_0 \uplus \mathit{Qmap}_1, \\ & & & \quad (\mathit{last}_0 \times \mathit{first}_1) \uplus (\mathit{last}_1 \times \mathit{first}_0) \uplus W_0 \uplus W_1)]_{\cong} \\ & & \textbf{end} & \end{array}$$

$$\begin{array}{lcll} C_{*,RFA'}([R]_{\cong}) & = & \textbf{let} & (Q, V, \mathit{follow}, \mathit{first}, \mathit{last}, \mathit{null}, \mathit{Qmap}, W) = R \\ & & \textbf{in} & \\ & & & [(Q, V, \mathit{follow} \uplus W, \mathit{first}, \mathit{last}, \mathit{true}, \mathit{Qmap}, \emptyset)]_{\cong} \\ & & \textbf{end} & \end{array}$$

$$C_{+,RFA'}([R]_\cong) = \textbf{let } (Q, V, follow, first, last, null, Qmap, W) = R \textbf{ in } [(Q, V, follow \uplus W, first, last, null, Qmap, \emptyset)]_\cong \textbf{ end}$$

$$C_{?,RFA'}([R]_\cong) = \textbf{let } (Q, V, follow, first, last, null, Qmap, W) = R \textbf{ in } [(Q, V, follow, first, last, true, Qmap, W)]_\cong \textbf{ end}$$

These operators are symmetrical. The correctness of these operators is shown in Theorem B.3. □

Chang and Paige make additional running-time savings by computing the components of a RFA' only as needed in the operators $C_{*,RFA'}$ and $C_{+,RFA'}$. The running-time and space savings, along with implementation details are given in [Chan92, CP92].

The second solution also involves adding an eighth tuple element to *RFA*'s, giving RFA''.

Definition 4.78 (RFA''): We add an eighth component W to each *RFA*

$$(Q, V, follow, first, last, null, Qmap, W)$$

such that

$$W = follow \setminus (last \times first)$$

These modified *RFA*'s are denoted by RFA''. □

As before, the new operators follow directly from the above definition.

Construction 4.79 (Operators of the Σ-algebra of RFA''): As usual, the requirement for binary operators is that the representatives of the arguments are chosen such that they have disjoint state sets.

$$C_{\epsilon,RFA''} = [(\emptyset, V, \emptyset, \emptyset, \emptyset, true, \emptyset, \emptyset)]_\cong$$
$$C_{\emptyset,RFA''} = [(\emptyset, V, \emptyset, \emptyset, \emptyset, false, \emptyset, \emptyset)]_\cong$$

$$C_{a\in V,RFA''} = \textbf{let } q_0 \text{ be a new state } \textbf{in } [(\{q_0\}, V, \emptyset, \{q_0\}, \{q_0\}, false, \{(q_0, a)\}, \emptyset)]_\cong \textbf{ end}$$

$$\begin{array}{rcl} C_{\cdot,RFA''}([R_0]_\cong, [R_1]_\cong) & = & \textbf{let } (Q_0, V, follow_0, first_0, last_0, null_0, Qmap_0, W_0) = R_0 \\ & & \quad\; (Q_1, V, follow_1, first_1, last_1, null_1, Qmap_1, W_1) = R_1 \\ & & \textbf{in} \\ & & \quad \textbf{let } first' = first_0 \uplus \textbf{if } (null_0) \textbf{ then } first_1 \textbf{ else } \emptyset \textbf{ fi} \\ & & \quad\quad last' = last_1 \uplus \textbf{if } (null_1) \textbf{ then } last_0 \textbf{ else } \emptyset \textbf{ fi} \\ & & \quad\quad W' = \textbf{if } (null_1) \textbf{ then } W_0 \textbf{ else } follow_0 \textbf{ fi} \\ & & \quad\quad W'' = \textbf{if } (null_0) \textbf{ then } W_1 \textbf{ else } follow_1 \textbf{ fi} \\ & & \quad\quad W'' = \textbf{if } (null_0 \wedge null_1) \textbf{ then } \emptyset \textbf{ else } (last_0 \times first_1) \textbf{ fi} \\ & & \quad \textbf{in} \\ & & \quad\quad [(Q_0 \uplus Q_1, V, follow_0 \uplus follow_1 \uplus (last_0 \times first_1), \\ & & \quad\quad\quad first', last', null_0 \wedge null_1, Qmap_0 \uplus Qmap_1, \\ & & \quad\quad\quad W' \uplus W'' \uplus W''')]_\cong \\ & & \quad \textbf{end} \\ & & \textbf{end} \end{array}$$

$$
\begin{array}{lll}
C_{\cup,RFA''}([R_0]_{\cong},[R_1]_{\cong}) & = & \textbf{let} \quad (Q_0, V, \mathit{follow}_0, \mathit{first}_0, \mathit{last}_0, \mathit{null}_0, \mathit{Qmap}_0, W_0) = R_0 \\
& & \qquad\quad (Q_1, V, \mathit{follow}_1, \mathit{first}_1, \mathit{last}_1, \mathit{null}_1, \mathit{Qmap}_1, W_1) = R_1 \\
& & \textbf{in} \\
& & \qquad\quad [(Q_0 \uplus Q_1, V, \mathit{follow}_0 \uplus \mathit{follow}_1, \mathit{first}_0 \uplus \mathit{first}_1, \\
& & \qquad\qquad \mathit{last}_0 \uplus \mathit{last}_1, \mathit{null}_0 \vee \mathit{null}_1, \mathit{Qmap}_0 \uplus \mathit{Qmap}_1, \\
& & \qquad\qquad W_0 \uplus W_1)]_{\cong} \\
& & \textbf{end}
\end{array}
$$

$$
\begin{array}{lll}
C_{*,RFA''}([R]_{\cong}) & = & \textbf{let} \quad (Q, V, \mathit{follow}, \mathit{first}, \mathit{last}, \mathit{null}, \mathit{Qmap}, W) = R \\
& & \textbf{in} \\
& & \qquad [(Q, V, W \uplus (\mathit{last} \times \mathit{first}), \mathit{first}, \mathit{last}, \mathit{true}, \mathit{Qmap}, W)]_{\cong} \\
& & \textbf{end}
\end{array}
$$

$$
\begin{array}{lll}
C_{+,RFA''}([R]_{\cong}) & = & \textbf{let} \quad (Q, V, \mathit{follow}, \mathit{first}, \mathit{last}, \mathit{null}, \mathit{Qmap}, W) = R \\
& & \textbf{in} \\
& & \qquad [(Q, V, W \uplus (\mathit{last} \times \mathit{first}), \mathit{first}, \mathit{last}, \mathit{null}, \mathit{Qmap}, W)]_{\cong} \\
& & \textbf{end}
\end{array}
$$

$$
\begin{array}{lll}
C_{?,RFA''}([R]_{\cong}) & = & \textbf{let} \quad (Q, V, \mathit{follow}, \mathit{first}, \mathit{last}, \mathit{null}, \mathit{Qmap}, W) = R \\
& & \textbf{in} \\
& & \qquad [(Q, V, \mathit{follow}, \mathit{first}, \mathit{last}, \mathit{true}, \mathit{Qmap}, W)]_{\cong} \\
& & \textbf{end}
\end{array}
$$

These operators are symmetrical. Their correctness is shown in Theorem B.4. □

Definition 4.80 (Mapping $[RFA']_{\cong}$ and $[RFA'']_{\cong}$ to $[RFA]_{\cong}$): The mapping is $\bar{\pi}_8$. □

Remark 4.81: Although this construction does not appear in the literature, a related one does: Brüggemann-Klein describes a transformation on regular expressions which closely parallels the RFA'' operators. A regular expression E is first transformed into *star normal-form* $E^{\bullet}$; the RFA image of $E^{\bullet}$ has similar properties to the RFA'' image of E. The details of the star normal-form transformations (and the running time improvements resulting from them) are described in [B-K93a]. □

5 The Myhill-Nerode, Brzozowski and DeRemer constructions

In this section, we explore a *DFA* construction method — due to Myhill and Nerode — from which we derive Brzozowski's construction. First, we make some observations about determinism in finite automata.

Recall a property of weakly deterministic automata — Property 2.28. Given that the set of left languages (of the states) in a *DFA* are disjoint, we will be exploring methods of computing a set of left languages to construct an automaton.

Definition 5.1 (Left languages of a *DFA*): Define the left languages of a *DFA* as:

$$\overleftarrow{\mathcal{L}}(Q,V,T,E,S,F) = \{\overleftarrow{\mathcal{L}}(q) : q \in Q\}$$

□

Since the elements of this set are pairwise disjoint (by Property 2.28), we can also view it as the (finite) set of equivalence classes of some equivalence relation on V^*. There are two potential problems with this:

- In the case that an $M \in DFA$ is not *Complete* then $\overleftarrow{\mathcal{L}}(M)$ is a partial partition of V^*. (This follows from Property 2.16.) To make the definitions in this section easier to present, we restrict ourselves to *Complete DFA*'s.
- It may be that $\emptyset$ is a left language of some state — corresponding to a start-unreachable state. In this section, we will not be interested in *DFA*'s with start-unreachable states.

Since $\mathcal{L}_{FA}(Q,V,T,E,S,F) = (\cup\ f : f \in F : \overleftarrow{\mathcal{L}}(f))$ (see Definition 2.12) we also note that the language of an automaton M is the union of some of the equivalence classes in $\overleftarrow{\mathcal{L}}(M)$.

Definition 5.2 (Right invariance of an equivalence relation): An equivalence relation E on V^* is right-invariant if and only if

$$(\forall\ u,a : u \in V^* \wedge a \in V : (\exists\ v : v \in V^* : [u]_E \cdot \{a\} \subseteq [v]_E))$$

□

Property 5.3 (Right invariance of an equivalence relation): Sometimes right invariance of equivalence relation E on V^* is given as

$$(\forall\ u,z : u \in V^* \wedge z \in V^* : (\exists\ v : v \in V^* : [u]_E \cdot \{z\} \subseteq [v]_E))$$

This is equivalent to the definition given above (by induction on the length of $z \in V^*$). □

We can now formulate an important property of $\overleftarrow{\mathcal{L}}(M)$, the partition of V^*.

Property 5.4 (Right invariance of a partition of V^*): Partition $\overleftarrow{\mathcal{L}}(Q,V,T,E,S,F)$ is right-invariant if and only if

$$(\forall\ p,a : p \in Q \wedge a \in V : (\exists\ q : q \in Q : \overleftarrow{\mathcal{L}}(p) \cdot \{a\} \subseteq \overleftarrow{\mathcal{L}}(q)))$$

□

Remark 5.5: It should be clear that for all $M \in DFA$ such that *Complete*(M), $\overleftarrow{\mathcal{L}}(M)$ is right-invariant; this follows since for all states p, (and transition relation $T \in Q \times V \longrightarrow Q$, since $M \in DFA$) (and $a \in V$):

$$\overleftarrow{\mathcal{L}}(p) \cdot \{a\} \subseteq \overleftarrow{\mathcal{L}}(T(p,a))$$

□

Remark 5.6: Had we been considering non-*Complete* *DFA*'s, we would not have a partition on V^*; right invariance could still be defined (for partial partition $\overleftarrow{\mathcal{L}}(Q, V, T, E, S, F)$) as:

$$(\forall\ p, a : p \in Q \wedge a \in V \wedge T^*(p, a) \neq \emptyset : (\exists\ q : q \in Q : \overleftarrow{\mathcal{L}}(p) \cdot \{a\} \subseteq \overleftarrow{\mathcal{L}}(q)))$$

We will not be using this definition. We give it to point out that the techniques of this section are not also usable in constructing non-*Complete* *DFA*'s (as Brzozowski demonstrated in [Brzo64]). □

5.1 The Myhill-Nerode construction

Before considering how to construct finite automata, we first present the Myhill-Nerode theorem. A good text book introduction to the theorem is [HU79].

Theorem 5.7 (Myhill-Nerode): The Myhill-Nerode theorem states that the following statements are equivalent [Myhi57, Nero58, RS59, HU79]:

1. L is a regular language.
2. L is the union of some of the equivalence classes of a right-invariant equivalence relation (on V^*) of finite index.
3. Let R_L be the right-invariant equivalence relation defined by

   $$(x, y) \in R_L \equiv (\forall\ z : z \in V^* : (xz \in L) \equiv (yz \in L))$$

   Relation R_L is of finite index.

Proof:
A following proof is given in [HU79, Theorem 3.9].

(1) ⇒ (2): Assume L is accepted by $M \in DFA$ such that $Complete(M)$. Let E be the equivalence relation corresponding to $\overleftarrow{\mathcal{L}}(M)$. $\sharp E$ is finite, and $L = (\cup\ f : f \in F : \overleftarrow{\mathcal{L}}(f))$. (See Definition A.8 for the definition of $\sharp$.)

(2) ⇒ (3): We show that for an equivalence relation E satisfying (2) that $E \subseteq R_L$. (Here $\subseteq$ denotes equivalence relation refinement, see Definition A.10.) We start the derivation using the right invariance property of E (Property 5.4, written slightly differently):

$$\begin{array}{ll}
 & (\forall\ u : u \in V^* : (\forall\ w : w \in V^* : (\exists\ v : v \in V^* : ([u]_E \cdot \{w\}) \subseteq [v]_E))) \\
\Rightarrow & \quad \{\text{Assumption that } E \text{ satisfies (2), for all } v \in V^*\text{: } ([v]_E \subseteq L) \vee ([v]_E \cap L = \emptyset)\} \\
 & (\forall\ u : u \in V^* : (\forall\ w : w \in V^* : (([u]_E \cdot \{w\}) \subseteq L) \vee (([u]_E \cdot \{w\}) \cap L = \emptyset))) \\
\Rightarrow & \quad \{\text{Definition of } R_L\} \\
 & (\forall\ u : u \in V^* : (\exists\ v : v \in V^* : [u]_E \subseteq [v]_{R_L})) \\
\equiv & \quad \{\text{Definition of refinement } (\sqsubseteq) \text{ — Definition A.11}\} \\
 & [V^*]_E \sqsubseteq [V^*]_{R_L} \\
\equiv & \quad \{\text{Definition of refinement } (\subseteq) \text{ — Definition A.10}\} \\
 & E \subseteq R_L \\
\Rightarrow & \quad \{\text{Property of refinement — Property A.12}\} \\
 & \sharp E \geq \sharp R_L
\end{array}$$

It follows that since $\sharp E$ is finite, so is $\sharp R_L$.

(3) ⇒ (1): We can construct the following *Complete DFA* (from R_L) accepting L:

$$\begin{array}{ll}\textbf{let} & T([w]_{R_L}, a) = \{[wa]_{R_L}\} \\ & F = \{[w]_{R_L} : w \in L\} \\ \textbf{in} & \\ & ([V^*]_{R_L}, V, T, \emptyset, \{[\epsilon]_{R_L}\}, F) \\ \textbf{end} & \end{array}$$

It follows that L is a regular language.

□

Property 5.8 (Index of equivalence relation E): Given $L \in \mathcal{L}_{\text{reg}}$, $M \in$ *DFA* accepting L, and E (satisfying statement 2 and constructed as in the proof of (1) ⇒ (2) of the Myhill-Nerode theorem) then $\sharp E \le |M|$ (since $\sharp E = \sharp \overleftarrow{\mathcal{L}}(M)$ by definition of E and Det'). □

Property 5.9 (Uniqueness and minimality of equivalence relation R_L): Of all equivalence relations satisfying statement 2 of the Myhill-Nerode theorem, R_L is the unique minimal one. This follows from the fact that all others are refinements of R_L (see the proof of (2) ⇒ (3)). □

The theorem does not say much about how to find equivalence relations satisfying statement 2, other than providing a definition of the unique minimal one, R_L.

We can formalize statement 2 of the Myhill-Nerode theorem:

Definition 5.10 (Predicate MN): For regular language L and equivalence relation E (on V^*) $MN(L, E)$ is equivalent to

- $\sharp E$ is finite,
- $L = (\cup\ v : v \in L : [v]_E)$, and
- E is right-invariant.

□

Note that $MN(L, R_L)$.

The *DFA* construction given in the (3) ⇒ (1) proof can be used with other right-invariant equivalence relations.

Construction 5.11 (Myhill-Nerode): Given a language L and right-invariant equivalence relation E such that $MN(L, E)$ we can construct an automaton accepting L using the function

$$\begin{array}{lcll} MNconstr(L, E) & = & \textbf{let} & T([w]_E, a) = \{[wa]_E\} \\ & & & F = \{[w]_E : w \in L\} \\ & & \textbf{in} & \\ & & & ([V^*]_E, V, T, \emptyset, \{[\epsilon]_E\}, F) \\ & & \textbf{end} & \end{array}$$

This construction has the following properties:

- The definition is independent of the choice of representatives of the equivalence classes of E.
- By inspection we can see that the *FA* constructed by *MNconstr* is a *Complete DFA*.
- For any state $U \in [V^*]_E$ we have $\overleftarrow{\mathcal{L}}(U) = U$.
- All states in the constructed automaton are start-reachable.
- The number of states is $\sharp E$.
- The construction satisfies the property

$$(\forall\ L, E : MN(L, E) : \mathcal{L}_{FA}(MNconstr(L, E)) = L)$$

□

5.2 The minimal equivalence relation R_L

The only relation (corresponding to $L \in \mathcal{L}_{\text{reg}}$) that Myhill and Nerode actually defined was R_L. This relation is particularly important, being the one of minimal index.

Theorem 5.12 (Unique minimal *DFA*): Given $L \in \mathcal{L}_{\text{reg}}$, $M = \mathit{MNconstr}(L, R_L)$ is the unique minimal *Complete DFA* accepting L; that is, $\mathit{Min}_C(M)$.
Proof:
Assume there exists $M' \in \mathit{DFA}$ such that $\mathit{Complete}(M')$, $\mathcal{L}_{FA}(M') = L$ and $|M'| \leq |M|$. From the proof of (2) $\Rightarrow$ (3) (and Property 5.8), $\sharp R_L \leq \sharp \overleftarrow{\mathcal{L}}(M')$. In summary, $\sharp R_L \leq \sharp \overleftarrow{\mathcal{L}}(M') \leq |M'| \leq |M| = \sharp R_L$, and so (by Property A.12) $|M'| = |M|$, $E = R_L$ and $M' \cong M$. $\square$

Property 5.13 (Reformulating R_L): We can rewrite the definition of R_L using derivatives (see Definition A.15) as follows:

$$\begin{array}{ll} & (x, y) \in R_L \\ \equiv & \quad \{\text{Property of derivatives and definition of } R_L\} \\ & (\forall\, z : z \in V^* : (z \in x^{-1}L) \equiv (z \in y^{-1}L)) \\ \equiv & \quad \{\text{Definition of} = \text{on languages}\} \\ & x^{-1}L = y^{-1}L \end{array}$$

$\square$

We could combine this definition of R_L with *MNconstr* to get a minimal *DFA* construction. Such a function would have a clumsy definition, and therefore we explore some encoding tricks.

5.2.1 Encoding R_L

An encoding trick is hinted at by Property 5.13: every equivalence class $[w]_{R_L}$ of R_L can be characterized by the language $w^{-1}L$.

Definition 5.14 (Derivative set of a language): We define the set of derivatives of language L as

$$\mathit{deriv}(L) = \{v^{-1}L : v \in V^*\}$$

$\square$

We have the following theorem relating to *deriv*

Theorem 5.15 (Finiteness of derivatives): If $L \in \mathcal{L}_{\text{reg}}$ then $|\mathit{deriv}(L)|$ is finite.
Proof:
$\sharp R_L$ is finite (from the Myhill-Nerode theorem), and since $|\mathit{deriv}(L)| = \sharp R_L$, $|\mathit{deriv}(L)|$ is also finite. $\square$

This theorem has also been given by Brzozowski [Brzo64]. His proof is, however, somewhat more complicated, and is by induction on the structure of language L.

Definition 5.16 (Encoding an equivalence class): We define a derivative encoding function (for a given $L \in \mathcal{L}_{\text{reg}}$) $\mathit{encderiv}_L \in [V^*]_{R_L} \longrightarrow \mathit{deriv}(L)$ as

$$\mathit{encderiv}_L([w]_{R_L}) = w^{-1}L$$

This function has inverse $\mathit{encderiv}_L^{-1}(v^{-1}L) = [v]_{R_L}$. Both of these functions are independent of the choice of representative of equivalence class of R_L. $\square$

Remark 5.17: In Construction 5.11 (with R_L as the equivalence class parameter) the equivalence classes of R_L are the left languages of the states of a *DFA* constructed from R_L. The function $encderiv_L$ maps these left languages (equivalence classes) to their corresponding right languages (the derivatives). The right language ($encderiv_L(U)$) of a particular equivalence class $U \in [V^*]_{R_L}$ is called the *continuation* of U (in language L) by Berry and Sethi [BS86]. □

Property 5.18 (Derivatives and function $encderiv_L$): Note that (for $a \in V$, $w \in V^*$):

- $encderiv_L([\epsilon]_{R_L}) = \epsilon^{-1}L = L$,
- $encderiv_L([wa]_{R_L}) = (wa)^{-1}L = a^{-1}(w^{-1}L)$.

These properties follow from Property A.16, and the definition of $encderiv_L$. □

Noting the form of function *MNconstr*, we use the encoding (function $encderiv_L$) to obtain construction $MNmin \in \mathcal{L}_{\text{reg}} \longrightarrow DFA$.

Construction 5.19 (*MNmin*): Combining *MNconstr* (Construction 5.11) with $encderiv_L$ (and its inverse) gives construction $MNmin \in \mathcal{L}_{\text{reg}} \longrightarrow DFA$:

$$
\begin{array}{lcll}
MNmin(L) & = & \textbf{let} & T(w^{-1}L, a) = \{a^{-1}(w^{-1}L)\} \\
 & & & F = \{w^{-1}L : \epsilon \in w^{-1}L\} \\
 & & \textbf{in} & \\
 & & & (deriv(L), V, T, \emptyset, \{L\}, F) \\
 & & \textbf{end} &
\end{array}
$$

Since *MNmin* is defined using *MNconstr*, the properties are similar:

- By inspection we can see that the *FA* constructed by *MNmin* is a *Complete DFA*.
- For any state $U \in deriv(L)$ we have $\overrightarrow{\mathcal{L}}(U) = U$.
- All states in the constructed automaton are start-reachable.
- The only state that is not final-reachable is $\emptyset$. The state $\emptyset$ exists in automaton $MNmin(L)$ if and only if $L \neq V^*$. It follows that we can remove the sink state $\emptyset$, to obtain a (possibly) non-*Complete DFA* with only useful states.
- The constructed *DFA* is the unique (up to isomorphism) minimal *Complete DFA* accepting L (since R_L is implicit in the definition).
- The construction satisfies the property

$$(\forall\ L : L \in \mathcal{L}_{\text{reg}} : \mathcal{L}_{FA}(MNmin(L)) = L)$$

□

Example 5.20 (*MNmin* construction): We construct the minimal *Complete DFA* corresponding to the regular language $\{\epsilon, a\}\{b\}^*$, denoted by regular expression $(a \cup \epsilon)b^*$ (the regular expression from Example 3.15).

After some calculation (using Property A.17):

$$
\begin{array}{rcl}
a^{-1}(\{\epsilon, a\}\{b\}^*) & = & (a^{-1}\{\epsilon, a\})\{b\}^* \cup a^{-1}\{b\}^* \\
 & = & \{\epsilon\}\{b\}^* \cup (a^{-1}\{b\})\{b\}^* \\
 & = & \{b\}^* \\
b^{-1}(\{\epsilon, a\}\{b\}^*) & = & (b^{-1}\{\epsilon, a\})\{b\}^* \cup b^{-1}\{b\}^* \\
 & = & \emptyset\{b\}^* \cup (b^{-1}\{b\})\{b\}^* \\
 & = & \{b\}^*
\end{array}
$$

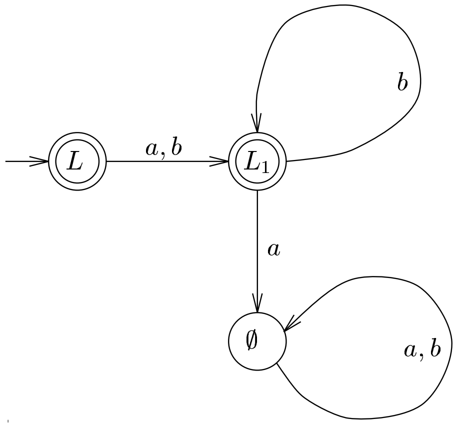


Figure 8: The *DFA* *MNmin*$(\{\epsilon, a\}\{b\}^*)$.

$$\begin{aligned}
a^{-1}(\{b\}^*) &= (a^{-1}\{b\})\{b\}^* \\
&= \emptyset\{b\}^* \\
&= \emptyset \\
b^{-1}(\{b\}^*) &= (b^{-1}\{b\})\{b\}^* \\
&= \{b\}^* \\
a^{-1}\emptyset &= \emptyset \\
b^{-1}\emptyset &= \emptyset
\end{aligned}$$

we determine that the three derivatives are: $L = \{\epsilon, a\}\{b\}^*$, $L_1 = \{b\}^*$, and $\emptyset$. The state graph is shown in Figure 8. □

5.3 The Brzozowski construction

We now concentrate on constructing *DFA*'s from extended regular expressions, as opposed to constructing them from regular languages. In Property A.17, a method is given for computing a derivative of a regular language (based upon the structure of the language). Being able to compute derivatives in this way also provides us with a definition of derivatives of extended regular expressions (*ERE*'s). Extended regular expressions were defined in Definition 4.53.

Remark 5.21: The Σ-algebra definition of regular expressions is not used in this section as the algebraic structure is not needed. Regular expressions are used only as syntactic objects, denoting regular languages. □

Remark 5.22: The remaining constructions in this section do not necessarily depend on extended regular expressions (normal regular expressions can also be used). They are introduced because some regular languages have more succinct descriptions as *ERE*'s than as *RE*'s. □

Definition 5.23 (Derivatives of *ERE*'s): Assuming $a \in V$ and $E, E_0, E_1 \in ERE$

$$\begin{aligned}
a^{-1}\emptyset &= \emptyset \\
a^{-1}\epsilon &= \emptyset \\
a^{-1}b &= \textbf{if } (a = b) \textbf{ then } \epsilon \textbf{ else } \emptyset \textbf{ fi} \qquad \text{(for all } b \in V) \\
a^{-1}(E_0E_1) &= (a^{-1}E_0)E_1 \cup \textbf{if } (\epsilon \in \mathcal{L}_{ERE}(E_0)) \textbf{ then } a^{-1}E_1 \textbf{ else } \emptyset \textbf{ fi} \\
a^{-1}(E_0 \cup E_1) &= (a^{-1}E_0) \cup (a^{-1}E_1)
\end{aligned}$$

$$\begin{array}{rcl} a^{-1}(E^*) & = & (a^{-1}E)E^* \\ a^{-1}(E^+) & = & (a^{-1}E)E^* \\ a^{-1}(E^?) & = & a^{-1}E \\ a^{-1}(E_0 \cap E_1) & = & (a^{-1}E_0) \cap (a^{-1}E_1) \\ a^{-1}(\neg E) & = & \neg(a^{-1}E) \end{array}$$

$\square$

Property 5.24 (Derivatives of *ERE*'s): By inspecting the definition of derivatives of *ERE*'s, we can verify that (for $a \in V, E \in ERE$) $a^{-1}\mathcal{L}_{ERE}(E) = \mathcal{L}_{ERE}(a^{-1}E)$. $\square$

Remark 5.25: Given equivalence relation $\doteq$ (equivalence of regular expressions, extended to *ERE*), an $E \in ERE$ will have a finite number of derivatives. More formally, $|\{[w^{-1}E]_{\doteq} : w \in V^*\}|$ is finite. Brzozowski gave the same result by induction on the structure of a *ERE* in [Brzo64, Theorem 4.3a]. $\square$

Definition 5.26 (Similarity ($\sim$) of regular expressions): Similarity (written $\sim$) is an equivalence relation on *ERE*'s. Two *ERE*'s are similar if and only if they are identical or one can be transformed into the other using the following rules:

1. $E_0 \cup E_1 = E_1 \cup E_0$ (commutativity of $\cup$),
2. $E_0 \cup (E_1 \cup E_2) = (E_0 \cup E_1) \cup E_2$ (associativity of $\cup$), and
3. $E \cup E = E$ (idempotence of $\cup$).

$\square$

Property 5.27 (Similarity): $\sim \subseteq \doteq$; that is, $\sim$ is a refinement of $\doteq$. $\square$

Definition 5.28 (The derivatives of an *ERE*): Function $deriv_{ERE} \in ERE \longrightarrow \mathcal{P}([ERE]_\sim)$ is defined as

$$deriv_{ERE}(E) = [\{v^{-1}E : v \in V^*\}]_\sim = \{[v^{-1}E]_\sim : v \in V^*\}$$

$\square$

Before proving that $deriv_{ERE}(E)$ is finite (for all $E \in ERE$), we need the following proposition.

Proposition 5.29 (Similarity equivalence class of a union *ERE*): Assume a finite set $H \subseteq ERE$ and a fully parenthesized regular expression

$$J = (\cdots(((h_1 \cup h_2) \cup h_3) \cup h_4) \cdots \cup h_k)$$

where (for $1 \le i \le k$) $h_i \in H$; each h_i is called a *term* of J. Using $\sim$ we can always find a similar (and, of course, equivalent) regular expression K, where K is the union of at most $|H|$ terms of J. This is because the rules defining $\sim$ can be used to reassociate and commute the terms of J (to place identical terms adjacent to one another), while the idempotence rule of $\sim$ can be used to remove identical terms. $\square$

Proposition 5.30 (Similarity): Given a finite set $H \subseteq ERE$, the set of all non-similar *ERE*'s that are unions of terms $h_i \in H$ is finite. $\square$

Theorem 5.31 (Finiteness of derivatives under similarity): For all $E \in ERE$, $|deriv_{ERE}(E)|$ is finite.
Proof:
This proof is similar to the one given in Brzozowski's original paper [Brzo64, Theorem 5.2]. The proof is by induction on the number of operators in $E \in ERE$.

Basis: The theorem is true for each of the constants: $deriv_{ERE}(\epsilon) = \{[\epsilon]_\sim, [\emptyset]_\sim\}$, $deriv_{ERE}(\emptyset) = \{[\emptyset]_\sim\}$, and $deriv_{ERE}(a) = \{[a]_\sim, [\epsilon]_\sim, [\emptyset]_\sim\}$ (for $a \in V$).

Induction hypothesis: Assume that $|deriv_{ERE}(E)|$ is finite for all $E \in ERE$ where E has fewer than k operators.

Induction step: Assume $E \in ERE$ has k operators. We use case analysis to deal with the possible forms of E:

$E = E_0 \cup E_1$**:** It is possible to show that

$$\{[w^{-1}(E_0 \cup E_1)]_\sim : w \in V^*\} = \{[w^{-1}E_0 \cup w^{-1}E_1]_\sim : w \in V^*\}$$

By the induction hypothesis, $|deriv_{ERE}(E_0)|$ and $|deriv_{ERE}(E_1)|$ are finite, and so is $|deriv_{ERE}(E)|$.

$E = E_0 \cap E_1$ **or** $E = E_0^?$ **or** $E = \neg E_0$**:** An argument similar to that for $\cup$ applies to these cases.

$E = E_0 \cdot E_1$**:** In order to analyze $|deriv_{ERE}(E_0 \cdot E_1)|$, we consider a particular $w^{-1}(E_0 \cdot E_1)$ (for $w \in V^*$). Let $w = a_1 \cdots a_n$ where each $a_i \in V$. Writing out $w^{-1}(E_0 \cdot E_1)$ we get

$$\begin{aligned} & w^{-1}(E_0 \cdot E_1) \\ = \quad & (a_2 \cdots a_n)^{-1}((a_1^{-1}E_0)E_1 \cup \textbf{if } (\epsilon \in \mathcal{L}_{ERE}(E_0)) \textbf{ then } a_1^{-1}E_1 \textbf{ else } \emptyset \textbf{ fi}) \end{aligned}$$

Had we been able to continue this rewriting, we would see that $[w^{-1}(E_0 \cdot E_1)]_\sim$ is equal to

$$[(w^{-1}E_0) \cdot E_1 \cup (\cup\ u, v : uv = w : \textbf{if } (\epsilon \in \mathcal{L}_{ERE}(u^{-1}E_0)) \textbf{ then } v^{-1}E_1 \textbf{ else } \emptyset \textbf{ fi})]_\sim$$

That is, $w^{-1}(E_0 \cdot E_1)$ is the union of a set of terms, one of which is $(w^{-1}E_0) \cdot E_1$, and the remaining ones are either a derivative of E_1, or $\emptyset \in ERE$. By the induction hypothesis (the set of derivatives of E_1 is finite) the set of possible terms is finite. It follows from Propositions 5.29 and 5.30 that $|deriv_{ERE}(E)|$ is finite.

$E = E_0^*$ **or** $E = E_0^+$**:** As in the $E_0 \cdot E_1$ case, we could write out $w^{-1}(E_0^*)$ for a particular $w \in V^*$. If we do this, we see that it is the union of terms, each of which is a derivative of E_0 concatenated with E_0^*. The set of possible terms is finite — by the induction hypothesis. Again, it follows from Propositions 5.29 and 5.30 that $|deriv_{ERE}(E)|$ is finite.

□

Remark 5.32: Unfortunately, using similarity (in computing derivatives) may yield more derivatives than recognizing equivalence ($\doteq$) of derivatives (as shown in Example 5.37). The rules defining similarity can be augmented with others to decrease the redundancy of the derivatives (and therefore the size of the constructed *DFA*). Any equivalence relation G such that $\sim \subseteq G \subseteq \doteq$ is usable for this. Examples of additional rules are (for $E_0, E_1, E_2 \in ERE$):

1. $E_0 \cdot \emptyset = \emptyset$ ($\emptyset$ is the zero of concatenation),
2. $E_0 \cup \emptyset = E_0$ ($\emptyset$ is the unit of $\cup$),
3. $E_0 \cdot \epsilon = E_0$ (ϵ is the unit of concatenation),
4. $\emptyset^* = \epsilon$ (a property of $*$),
5. $E_0 \cdot (E_1 \cup E_2) = E_0 \cdot E_1 \cup E_0 \cdot E_2$ ($\cdot$ distributes over $\cup$),
6. $E_0 \cap \emptyset = \emptyset$ ($\emptyset$ is the zero of $\cap$),
7. $E_0 \cap E_1 = E_1 \cap E_0$ (commutativity of $\cap$),

8. $E_0 \cap (E_1 \cap E_2) = (E_0 \cap E_1) \cap E_2$ (associativity of $\cap$).

9. $E \cap E = E$ (idempotence of $\cap$).

□

There is a property of similarity that will be needed to present the Brzozowski construction.

Definition 5.33 (Derivative of a similarity equivalence class): For $E \in ERE$ and $a \in V$ we have $a^{-1}[E]_\sim = [a^{-1}E]_\sim$. This definition does not depend on the choice of representative of the equivalence class (under $\sim$). □

The Brzozowski construction is an encoding of *MNmin* to use *ERE*'s and equivalence classes of $\sim$.

Construction 5.34 (Brzozowski): Function $Brz \in ERE \longrightarrow DFA$ is defined as:

$$\begin{array}{rcll} Brz(E) & = & \textbf{let} & T([v^{-1}E]_\sim, a) = \{a^{-1}[v^{-1}E]_\sim\} \\ & & & F = \{[w^{-1}E]_\sim : \epsilon \in \mathcal{L}_{ERE}(w^{-1}E)\} \\ & & \textbf{in} & \\ & & & (deriv_{ERE}(E), V, T, \emptyset, \{[E]_\sim\}, F) \\ & & \textbf{end} & \end{array}$$

The properties of *Brz* correspond to those of *MNmin*:

- The construction is independent of the representatives of equivalence classes.
- By inspection we can see that *Brz* constructs *Complete DFA*'s.
- For any state $E' \in deriv_{ERE}(E)$ we have $\overrightarrow{\mathcal{L}}(E') = \mathcal{L}_{ERE}(E')$.
- All states in the constructed automaton are start-reachable.
- There may be a state that is not final-reachable; this sink state will exists if and only if $\mathcal{L}_{ERE}(E) \neq V^*$. The sink state corresponds to the derivative $\emptyset \in ERE$.
- The construction satisfies the property

$$(\forall\ E : E \in ERE : \mathcal{L}_{FA}(Brz(E)) = \mathcal{L}_{ERE}(E))$$

□

Remark 5.35: Any equivalence relation G (on *ERE*'s) such that $\sim \subseteq G \subseteq \doteq$ can be used in place of $\sim$ in Brzozowski's construction. □

Remark 5.36: In Brzozowski's original paper [Brzo64], the sink state (corresponding to derivative $\emptyset \in ERE$) was always omitted from the constructed *DFA*, producing a possibly non-*Complete DFA*. □

Example 5.37 (Brzozowski's construction): We construct a *Complete DFA* corresponding to regular expression $(a \cup \epsilon)b^*$ (the regular expression from Example 3.15). The derivatives are:

$$\begin{array}{rcl} a^{-1}((a \cup \epsilon)b^*) & = & (a^{-1}(a \cup \epsilon))b^* \cup a^{-1}(b^*) \\ & = & (a^{-1}a \cup a^{-1}\epsilon)b^* \cup (a^{-1}b)b^* \\ & = & (\epsilon \cup \emptyset)b^* \cup \emptyset b^* \\ b^{-1}((a \cup \epsilon)b^*) & = & (b^{-1}(a \cup \epsilon))b^* \cup b^{-1}(b^*) \\ & = & (b^{-1}a \cup b^{-1}\epsilon)b^* \cup (b^{-1}b)b^* \\ & = & (\emptyset \cup \emptyset)b^* \cup \epsilon b^* \end{array}$$

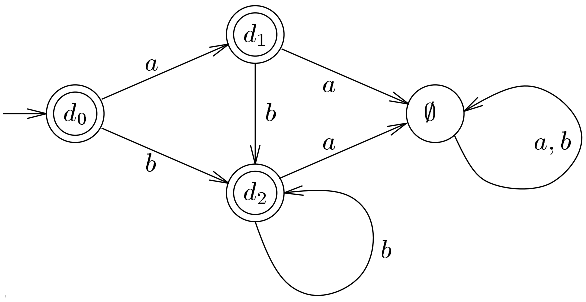


Figure 9: The *DFA* $Brz((a \cup \epsilon)b^*)$.

$$
\begin{array}{rcl}
 & \sim & \emptyset b^* \cup \epsilon b^* \\
a^{-1}((\epsilon \cup \emptyset)b^* \cup \emptyset b^*) & = & a^{-1}((\epsilon \cup \emptyset)b^*) \cup a^{-1}(\emptyset b^*) \\
 & = & ((a^{-1}(\epsilon \cup \emptyset))b^* \cup a^{-1}(b^*)) \cup (a^{-1}\emptyset)b^* \\
 & = & (((a^{-1}\epsilon \cup a^{-1}\emptyset)b^*) \cup (a^{-1}b)b^*) \cup \emptyset b^* \\
 & = & ((\emptyset \cup \emptyset)b^* \cup (\emptyset)b^*) \cup \emptyset b^* \\
 & \sim & \emptyset b^* \\
b^{-1}((\epsilon \cup \emptyset)b^* \cup \emptyset b^*) & = & b^{-1}((\epsilon \cup \emptyset)b^*) \cup b^{-1}(\emptyset b^*) \\
 & = & ((b^{-1}(\epsilon \cup \emptyset))b^* \cup (b^{-1}b)b^*) \cup (b^{-1}\emptyset)b^* \\
 & = & ((b^{-1}\epsilon \cup b^{-1}\emptyset)b^* \cup (\epsilon)b^*) \cup \emptyset b^* \\
 & = & ((\emptyset \cup \emptyset)b^* \cup \epsilon b^*) \cup \emptyset b^* \\
 & \sim & \emptyset b^* \cup \epsilon b^* \\
a^{-1}(\emptyset b^* \cup \epsilon b^*) & = & a^{-1}(\emptyset b^*) \cup a^{-1}(\epsilon b^*) \\
 & = & (a^{-1}\emptyset)b^* \cup ((a^{-1}\epsilon)b^* \cup (a^{-1}b)b^*) \\
 & \sim & \emptyset b^* \\
b^{-1}(\emptyset b^* \cup \epsilon b^*) & = & b^{-1}(\emptyset b^*) \cup b^{-1}(\epsilon b^*) \\
 & = & (b^{-1}\emptyset)b^* \cup ((b^{-1}\epsilon)b^* \cup (b^{-1}b)b^*) \\
 & \sim & \emptyset b^* \cup \epsilon b^* \\
a^{-1}(\emptyset b^*) & = & (a^{-1}\emptyset)b^* \\
 & = & \emptyset b^* \\
b^{-1}(\emptyset b^*) & = & (b^{-1}\emptyset)b^* \\
 & = & \emptyset b^*
\end{array}
$$

The four derivatives (under $\sim$) are: $d_0 = (a \cup \epsilon)b^*$, $d_1 = (\epsilon \cup \emptyset)b^* \cup \emptyset b^*$, $d_2 = \emptyset b^* \cup \epsilon b^*$, and $d_3 = \emptyset b^*$. The state graph is shown in Figure 9. Had we been able to recognize equivalence of *ERE*'s, we would have had a smaller *DFA* since $(\epsilon \cup \emptyset)b^* \cup \emptyset b^* \doteq \emptyset b^* \cup \epsilon b^*$, and we could have identified states d_1 and d_2. $\square$

5.3.1 Computing derivatives of an *ERE*

Brzozowski also shows [Brzo64] if $E \in ERE$ has n derivatives (including E, under any equivalence relation G such that $\sim \subseteq G \subseteq \doteq$) then they are all of the form $v^{-1}E$ where $|v| < n$ [Brzo64, Theorem 4.3b]. Also part of this theorem is if all derivatives (of E) with respect to strings of length not greater than n have been found, and no new ones are found with respect to strings of length $n+1$, then no new ones will be found with respect to strings of length greater than n. This useful property of derivatives (in fact a slightly stronger property) can be stated as follows:

Theorem 5.38 (Finding derivatives): For all $r \geq 0$

$$\begin{array}{ll} & \{[w^{-1}E]_\sim : w \in V^* \wedge |w| = r\} \subseteq \{[w^{-1}E]_\sim : w \in V^* \wedge |w| < r\} \\ \Rightarrow & \\ & \{[w^{-1}E]_\sim : w \in V^* \wedge |w| \geq r\} \subseteq \{[w^{-1}E]_\sim : w \in V^* \wedge |w| < r\} \end{array}$$

□

This gives the following algorithm (in the guarded commands of [Dijk76]) which computes the derivatives of a regular expression E ($D \subseteq [ERE]_\sim$ and $next \subseteq [ERE]_\sim$):

Algorithm 5.39:

$\{E \in ERE\}$
$D, next, k := \emptyset, \{[E]_\sim\}, 0;$
$\{\text{invariant: } D = \{[w^{-1}E]_\sim : w \in V^* \wedge |w| < k\} \wedge next = \{[w^{-1}E]_\sim : w \in V^* \wedge |w| = k\}\}$
do $next \not\subseteq D \longrightarrow$
  $D, next, k := D \cup next, \{a^{-1}F : a \in V \wedge F \in next\}, k + 1$
od$\{D = deriv_{ERE}(E)\}$

5.3.2 Extending derivatives

It is sometimes useful to extend derivatives to deal with additional operators: prefix closure and certain functions on languages. We now briefly give the definition of derivatives of regular languages (and thus regular expressions) with these operators.

The prefix closure of a language is defined as:

$$\mathbf{pref}(L) = \{u : u^{-1}L \neq \emptyset\}$$

and the derivative of a prefix closed language is:

$$a^{-1}(\mathbf{pref}(L)) = \mathbf{pref}(a^{-1}L)$$

For certain functions $f \in \mathcal{L}_{\text{reg}} \times \mathcal{L}_{\text{reg}} \longrightarrow \mathcal{L}_{\text{reg}}$, derivatives are defined as:

$$a^{-1}(f(L_0, L_1)) = f(a^{-1}L_0, a^{-1}L_1)$$

Some examples of such functions are $\cap$, $\cup$, asymmetrical difference, and symmetrical difference. For more on this see [Brzo64].

5.4 Relating the Brzozowski and Berry-Sethi constructions

It turns out that for *RRE*'s (recall from Definition 4.58 that an $E \in RRE$ is an RE such that each symbol of V occurs at most once in E), the Brzozowski construction (with sink state removal — as in Brzozowski's original paper — and a suitable encoding) and the Berry-Sethi construction produce isomorphic *DFA*'s. In this section, we consider only *RRE*'s. Berry and Sethi first presented this result in [BS86].

We will be using the following version of Brzozowski's construction (for $E \in RRE$), which does not introduce a sink state (the sink state is equivalent ($\doteq$) to $\emptyset \in RRE$ — its language under $\mathcal{L}_{RE}$ is $\emptyset$).

Construction 5.40 (Brzozowski — without sink state): Given $E \in RE$, the following constructs a *DFA* accepting $\mathcal{L}_{RE}(E)$:

$$\begin{array}{ll} \textbf{let} & Q = \{[w^{-1}E]_\sim : w \in V^* \wedge \mathcal{L}_{RE}(w^{-1}E) \neq \emptyset\} \\ & T([v^{-1}E]_\sim, a) = \textbf{if } (\mathcal{L}_{RE}(a^{-1}(v^{-1}E)) \neq \emptyset) \textbf{ then } \{a^{-1}[v^{-1}E]_\sim\} \textbf{ else } \emptyset \textbf{ fi} \\ & F = \{[w^{-1}E]_\sim : Null(w^{-1}E)\} \\ \textbf{in} & \\ & (Q, V, T, \emptyset, \{[E]_\sim\}, F) \\ \textbf{end} & \end{array}$$

In the **let** clause, the transition function has signature $T \in Q \times V \longrightarrow \mathcal{P}(Q)$. Recall from Definition 3.20 that $Null(E) \equiv \epsilon \in \mathcal{L}_{RE}(E)$. $\square$

For any $E \in RRE$, the only way that $\mathcal{L}_{RE}(b^{-1}((wa)^{-1}E)) \neq \emptyset$ is if $\mathcal{L}_{RE}((wa)^{-1}E) \neq \emptyset$ and a b can follow a wa in some string in $\mathcal{L}_{RE}(E)$.

In order to make the above construction practical, we explore the possibility of characterizing all of the derivatives of an $E \in RE$ (except for the derivative E itself) by the symbols occuring in E.

Definition 5.41 (Unambiguous regular expressions): An $E \in RE$ is said to be *unambiguous* if and only if, for all $a \in Occ(E)$ (function Occ is defined in Definition 4.57), the following set is a singleton set:

$$\{[(wa)^{-1}E]_\sim : w \in V^* \wedge \mathcal{L}_{RE}((wa)^{-1}E) \neq \emptyset\}$$

In other words, all derivatives of E by wa (for $w \in V^*$ and $a \in V$) are either equivalent to $\emptyset \in RRE$, or are similar to one another. $\square$

Remark 5.42: If an $E \in RE$ is unambiguous, its derivatives are either E or $\emptyset$, or can be characterized by an element of $Occ(E)$. $\square$

Remark 5.43: The regular expression $(a \cup a)$ is unambiguous, but is not an RRE. $\square$

Remark 5.44: Unambiguous regular expressions are also defined by Champarnaud [Cham93], although he characterizes them quite differently, and he does not make use of derivatives. Champarnaud calls such regular expressions *local*. $\square$

Theorem 5.45 (Characterizing derivatives of RRE's): For any $E \in RRE$, E is unambiguous. This theorem is also given by Berry and Sethi [BS86, Theorem 3.4].

Proof:

We proceed by induction on the number of operators in $E \in RRE$.

Basis: The theorem is trivially true for the RRE base cases ϵ and $\emptyset$ since $Occ(\epsilon) = Occ(\emptyset) = \emptyset$. It is also trivially true for the RRE $a \in V$.

Induction hypothesis: Assume that the theorem is true for any $E \in RRE$ with fewer than k operators.

Induction step: We now consider $E \in RRE$ with k operators. We now examine the possible structure of E (assuming $a \in Occ(E)$).

$E = E_0 \cup E_1$**:** Given $w \in V^*$ such that $\mathcal{L}_{RE}((wa)^{-1}E) \neq \emptyset$

$$(wa)^{-1}(E_0 \cup E_1) = ((wa)^{-1}E_0) \cup ((wa)^{-1}E_1)$$

Since $E \in RRE$, then either $a \in Occ(E_0)$ or $a \in Occ(E_1)$ (but not both). It follows that $(wa)^{-1}(E_0 \cup E_1)$ is similar to $(wa)^{-1}E_0 \cup \emptyset$ or similar to $(wa)^{-1}E_1 \cup \emptyset$ (but not both). The theorem then follows from the induction hypothesis.

$E = E_0 \cdot E_1$**:** From Theorem 5.31 we know that (for $w \in V^*, a \in V$) $[(wa)^{-1}(E_0 \cdot E_1)]_\sim$ is equal to:

$$[((wa)^{-1}E_0)E_1 \cup (\cup\ u, v : uva = wa : \textbf{if } (\epsilon \in \mathcal{L}_{ERE}(u^{-1}E_0)) \textbf{ then } (va)^{-1}E_1 \textbf{ else } \emptyset \textbf{ fi})]_\sim$$

Since $E \in RRE$, then either $a \in Occ(E_0)$ or $a \in Occ(E_1)$ (but not both). It follows, by an argument similar to the $E = E_0 \cup E_1$ case (above), that the theorem holds from the induction hypothesis.

$E = E_0^*$, $E = E_0^+$ **or** $E = E_0^?$**:** The argument for these cases proceeds similarly to the $E = E_0 \cdot E_1$ case. For more on this type of argument see Theorem 5.31.

□

Remark 5.46: The above theorem implies that each derivative of $E \in RRE$ (under similarity) is either $\emptyset$ (which we ignore since it corresponds to the sink state) or E, or it can be characterized by an $a \in Occ(E)$. □

Definition 5.47 (Encoding the derivatives of an RRE): Given the theorem above, $E \in RRE$, we can now give a partial encoding function enc_E (corresponding to the $E \in RRE$) from the derivatives of E to $Occ(E)$ such that (for $w \in V^*, a \in V$) $enc_E((wa)^{-1}E) = a$ when $\mathcal{L}_{RE}((wa)^{-1}E) \neq \emptyset$, and enc_E is undefined otherwise. Note that this function is not defined on E. □

The following property makes use of the definitions of *Null*, *Occ*, *First*, *Last*, and *Follow* (Definitions 3.20, 4.57, 4.60–4.64 respectively).

Property 5.48 (Functions *Follow*, *First*, and *Last*): The following properties will be used:

- $(a, b) \in Follow(E) \equiv (\exists\ w : w \in V^* : \mathcal{L}_{RE}(b^{-1}((wa)^{-1}E)) \neq \emptyset)$.
- $a \in First(E) \equiv \mathcal{L}_{RE}(a^{-1}E) \neq \emptyset$.
- $a \in Last(E) \equiv \mathcal{L}_{RE}(Ea^{-1}) \neq \emptyset \equiv (\exists\ w : w \in V^* : \epsilon \in \mathcal{L}_{RE}((wa)^{-1}E))$.

Derivatives on the right are mentioned in Definition A.15. □

We can rewrite our sink stateless version of Brzozowski's construction, using the above properties, to obtain the construction now following.

Construction 5.49 (Encoding Brzozowski for RRE's): We can now give our encoded version (using $Occ, First, Last, Null$, and $Follow$) of Construction 5.40, as $Brzenc \in RRE \longrightarrow DFA$:

$$
\begin{array}{lcl}
Brzenc(E) & = & \textbf{let} \quad s \text{ be a new state (characterizing } E \in RRE) \\
& & \textbf{in} \\
& & \quad\quad \textbf{let} \quad T = \{(a, b, b) : (a, b) \in Follow(E)\} \\
& & \quad\quad\quad\quad\; T' = \{(s, a, a) : a \in First(E)\} \\
& & \quad\quad\quad\quad\; F = Last(E) \cup \textbf{if } (Null(E)) \textbf{ then } \{s\} \textbf{ else } \emptyset \textbf{ fi} \\
& & \quad\quad \textbf{in} \\
& & \quad\quad\quad\quad\; (Occ(E) \cup \{s\}, V, T \cup T', \emptyset, \{s\}, F) \\
& & \quad\quad \textbf{end} \\
& & \textbf{end}
\end{array}
$$

□

Remark 5.50: Using the set $Occ(E)$ as the set of states can yield a DFA with start-unreachable states. For example, in the DFA $Brzenc(\emptyset \cdot a)$, we have start-unreachable state a. □

Remark 5.51: By inspection we see that, for all $E \in RRE$, $Brzenc(E) \cong BSenc(E)$ (Construction 4.70). It follows from Remark 4.73, that for $E \in RRE$, $[Brz(E)]_{\cong} = MYG(E)$. □

Remark 5.52: Finally, we note that the construction $Brzenc$ produces a correct DFA for any $E \in RE$ such that E is unambiguous. That is, $E \in RRE$ is not required. This property is not noted in the literature. This follows from Definition 5.41 and the definition of $Brzenc$. □

5.5 Towards DeRemer's construction

In this subsection, we consider several more constructions based upon the *MNconstr* and *MNmin* constructions (Constructions 5.11 and 5.19). The idea is to characterize the derivatives of a regular expression by so-called dotted regular expressions. We only consider constructing a DFA from an RE, as opposed to an ERE. Since some of the proofs are tedious to present, we give this construction in an informal manner.

We begin by introducing *dotted regular expressions*, which are essentially regular expressions with a dot ($\bullet$) appearing in each of them.

Remark 5.53: We will be characterizing derivatives, not equivalence classes (as is required for *MNconstr*). Using dotted *RE*'s, it is considerably easier to characterize the derivatives than the equivalence classes. □

The dot should not be confused with the concatenation dot, the star normal-form dot of Brüggemann-Klein (presented in Section 4.5), or the bullet used in typesetting lists.

Definition 5.54 (Dotted regular expressions, their languages, and *undot*): We recursively define dotted regular expressions (*DRE*'s), function $\mathcal{R} \in DRE \longrightarrow \mathcal{P}(V^*)$, and function $undot \in DRE \longrightarrow RE$. Function $\mathcal{R}$ maps *DRE*'s to the (regular) language to the right of the dot, and function *undot* removes the dot in a *DRE*.

1. If $E \in RE$ then
   (a) $\bullet E \in DRE$, $\mathcal{R}(\bullet E) = \mathcal{L}_{RE}(E)$, and $undot(\bullet E) = E$;
   (b) $E\bullet \in DRE$, $\mathcal{R}(E\bullet) = \{\epsilon\}$, and $undot(E\bullet) = E$.
2. If $E \in RE$ and $D \in DRE$ then
   (a) $E \cup D \in DRE$, $D \cup E \in DRE$, $\mathcal{R}(E \cup D) = \mathcal{R}(D \cup E) = \mathcal{R}(D)$, $undot(E \cup D) = E \cup undot(D)$, and $undot(D \cup E) = undot(D) \cup E$;
   (b) $E \cdot D \in DRE$, $D \cdot E \in DRE$, $\mathcal{R}(E \cdot D) = \mathcal{R}(D)$, $\mathcal{R}(D \cdot E) = \mathcal{R}(D) \cdot \mathcal{L}_{RE}(E)$, $undot(E \cdot D) = E \cdot undot(D)$, and $undot(D \cdot E) = undot(D) \cdot E$;
   (c) $D^* \in DRE$, $\mathcal{R}(D^*) = \mathcal{R}(D) \cdot \mathcal{L}_{RE}(undot(D))^*$, and $undot(D^*) = undot(D)^*$;
   (d) $D^+ \in DRE$, $\mathcal{R}(D^+) = \mathcal{R}(D) \cdot \mathcal{L}_{RE}(undot(D))^*$, and $undot(D^+) = undot(D)^+$;
   (e) $D^? \in DRE$, $\mathcal{R}(D^?) = \mathcal{R}(D)$, and $undot(D^?) = undot(D)^?$.
3. Nothing else is a *DRE*.

A dotted regular expression is also known as an *item*, from LR parsing [Knut65]; we will frequently use this name. □

We also require a function mapping a regular expression to all of its dottings.

Definition 5.55 (Function *dots*): We define function $dots \in RE \longrightarrow \mathcal{P}(DRE)$ as follows:

$$dots(E) = \{D : D \in DRE \wedge undot(D) = E\}$$

□

Remark 5.56: For a given $E \in RE$, we will be using sets of items (elements of $\mathcal{P}(dots(E))$) to characterize the derivatives of $\mathcal{L}_{RE}(E)$ when constructing a *DFA* accepting $\mathcal{L}_{RE}(E)$. □

Property 5.57 (*dots*): For all $E \in RE$, $|dots(E)|$ is finite, and so is $|\mathcal{P}(dots(E))|$. □

We define an item set, and its language as follows:

Definition 5.58 (Item sets and their languages): An *item set* J is a subset of *DRE* such that:

$$(\exists\ E : E \in RE : J \subseteq dots(E))$$

IS denotes the set of all item sets. Essentially, an *IS* is a set of items, all of which are dottings of the same regular expression. We also extend *undot* to *IS*. □

Definition 5.59 (Language of an *IS*): The language of a $J \in IS$ is given by function $\mathcal{L}_{IS} \in IS \longrightarrow \mathcal{P}(V^*)$ defined as:

$$\mathcal{L}_{IS}(J) = (\cup\ I : I \in J : \mathcal{R}(I))$$

□

Remark 5.60: We will use item sets to characterize the derivatives of an *RE*. □

We now define derivatives of item sets.

Definition 5.61 (Derivative of an item set): Given $J \in IS$ and $a \in V$ we define $a^{-1}J$ to be the following set:

1. If $I \in J$ has a subexpression $\bullet a$ then I' is in $a^{-1}J$, where I' is the same as I with the subexpression $\bullet a$ replaced by $a\bullet$.
2. Nothing else is in $a^{-1}J$.

□

We can now define a special type of function, which we call a *closure function*.

Definition 5.62 (Closure functions): Any function $\mathcal{E} \in IS \longrightarrow IS$ can be used as a closure function, provided that

$$(\forall\ J : J \in IS : \mathcal{L}_{IS}(\mathcal{E}(J)) = \mathcal{L}_{IS}(J) \wedge \mathcal{E} \circ \mathcal{E}(J) = \mathcal{E}(J))$$

and

$$(\forall\ E, J : E \in RE \wedge J \subseteq dots(E) : \epsilon \in \mathcal{L}_{IS}(J) \equiv (E\bullet) \in \mathcal{E}(J))$$

and

$$(\forall\ J, a : J \in IS \wedge a \in V : \mathcal{L}_{IS}(a^{-1}\mathcal{E}(J)) = a^{-1}\mathcal{L}_{IS}(J))$$

□

We are now in a position to define our first closure function, and an auxiliary relation.

Definition 5.63 (Dot closure relation $\mathcal{D}$): We define a binary relation $\mathcal{D}$ on DRE. $\mathcal{D}$ is the smallest relation such that:

1. If $E, F \in RE$, then (here we use infix notation for relation $\mathcal{D}$):

$$\begin{array}{lll}
\bullet\epsilon & \mathcal{D} & \epsilon\bullet \\
\bullet(E \cdot F) & \mathcal{D} & (\bullet E) \cdot F \\
(E\bullet) \cdot F & \mathcal{D} & E \cdot (\bullet F) \\
E \cdot (F\bullet) & \mathcal{D} & (E \cdot F)\bullet \\
\bullet(E \cup F) & \mathcal{D} & (\bullet E) \cup F \\
\bullet(E \cup F) & \mathcal{D} & E \cup (\bullet F) \\
(E\bullet) \cup F & \mathcal{D} & (E \cup F)\bullet \\
E \cup (F\bullet) & \mathcal{D} & (E \cup F)\bullet \\
\bullet(E^*) & \mathcal{D} & (\bullet E)^* \\
\bullet(E^*) & \mathcal{D} & (E^*)\bullet \\
(E\bullet)^* & \mathcal{D} & (\bullet E)^* \\
(E\bullet)^* & \mathcal{D} & (E^*)\bullet \\
\bullet(E^+) & \mathcal{D} & (\bullet E)^+ \\
(E\bullet)^+ & \mathcal{D} & (\bullet E)^+ \\
(E\bullet)^+ & \mathcal{D} & (E^+)\bullet \\
\bullet(E^?) & \mathcal{D} & (\bullet E)^? \\
\bullet(E^?) & \mathcal{D} & (E^?)\bullet \\
(E\bullet)^? & \mathcal{D} & (E^?)\bullet
\end{array}$$

2. If $E \in RE$ and $D_0, D_1 \in DRE$ such that $(D_0, D_1) \in \mathcal{D}$, then:
   (a) $(E \cup D_0, E \cup D_1) \in \mathcal{D}$, $(D_0 \cup E, D_1 \cup E) \in \mathcal{D}$, $(E \cdot D_0, E \cdot D_1) \in \mathcal{D}$, and $(D_0 \cdot E, D_1 \cdot E) \in \mathcal{D}$.

(b) $(D_0^*, D_1^*) \in \mathcal{D}$, $(D_0^+, D_1^+) \in \mathcal{D}$, and $(D_0^?, D_1^?) \in \mathcal{D}$.

□

Definition 5.64 (Closure function $\mathcal{C}$): We define function $\mathcal{C} \in IS \longrightarrow IS$ as:

$$\mathcal{C}(J) = \mathcal{D}^*(J)$$

Function $\mathcal{C}$ satisfies the Definition 5.62, making it a closure function. □

Remark 5.65: The closure function $\mathcal{C}$ presented here is an extension (to deal with our definition of regular expressions) of the one usually given for LR parsing. □

Example 5.66 (Function $\mathcal{C}$): $\mathcal{C}(\{\bullet((a\cup\epsilon)b^*)\})$ is computed to be $\{\bullet((a\cup\epsilon)b^*), (\bullet(a\cup\epsilon))b^*, (\bullet a\cup\epsilon)b^*, (a\cup\bullet\epsilon)b^*, (a\cup\epsilon\bullet)b^*, ((a\cup\epsilon)\bullet)b^*, (a\cup\epsilon)(\bullet(b^*)), (a\cup\epsilon)(\bullet b)^*, (a\cup\epsilon)(b^*\bullet), ((a\cup\epsilon)b^*)\bullet\}$. □

In order to construct a *DFA* for an *RE*, we require a set of item sets which characterize the set of derivatives of a the regular expression. The following definition gives the necessary conditions.

Definition 5.67 (Derivative item set): Given $E \in RE$, the set $D \subseteq \mathcal{P}(dots(E))$ (that is, $D \subseteq IS$ and for each $J \in D$, $E = undot(J)$) characterizes (under some closure function $\mathcal{E}$) the derivatives of E if and only if:

$$\{w^{-1}\mathcal{L}_{RE}(E) : w \in V^*\} = \{\mathcal{L}_{IS}(J) : J \in D\}$$

and

$$(\forall\ J : J \in D : J = \mathcal{E}(J))$$

and

$$(\forall\ J, a : J \in D \wedge a \in V : \mathcal{E}(a^{-1}J) \in D)$$

We write this property $DIS(E, D, \mathcal{E})$. The set D is called a *derivative item set* for E. □

We are now in a position to modify Algorithm 5.39 to compute such a derivative item set (under some closure function $\mathcal{E}$), instead of a set of derivatives. In the following algorithm, $D, next \subseteq IS$.

Algorithm 5.68:

$\{E \in RE\}$
$D, next := \emptyset, \{\mathcal{E}(\{\bullet E\})\}$;
do $next \not\subseteq D \longrightarrow$
  $D, next := D \cup next, \{\mathcal{E}(a^{-1}I) : a \in V \wedge I \in next\}$
od
$\{DIS(E, D, \mathcal{E})\}$

This algorithm terminates since $|\mathcal{P}(dots(E))|$ is finite (Property 5.57). With the set D computed above, we can now construct a *DFA* accepting $\mathcal{L}_{RE}(E)$.

Construction 5.69 (Item set construction): Function $Iconstr \in RE \times \mathcal{P}(IS) \not\longrightarrow DFA$ takes a regular expression (E) and a derivative item set (D) for the *RE* (such that $DIS(E, D, \mathcal{C})$), and constructs a *DFA*:

$$\begin{array}{rcll} Iconstr(E, D) & = & \textbf{let} & T(J, a) = \{\mathcal{C}(a^{-1}J)\} \\ & & & S = \{\mathcal{C}(\{\bullet E\})\} \\ & & & F = \{J : J \in D \wedge E\bullet \in J\} \\ & & \textbf{in} & \\ & & & (D, V, T, \emptyset, S, F) \\ & & \textbf{end} & \end{array}$$

A *DFA* constructed with *Iconstr* has the following property:

$$(\forall\ E, D : DIS(E, D, \mathcal{C}) : \mathcal{L}_{FA}(Iconstr(E, D)) = \mathcal{L}_{RE}(E))$$

The *DFA* is also *Complete*. □

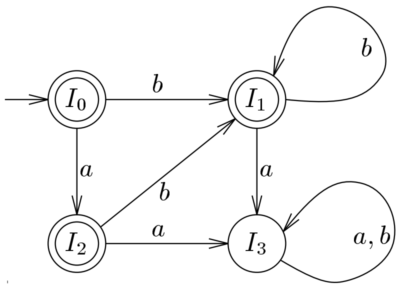


Figure 10: The *DFA* $Iconstr((a \cup \epsilon)b^*)$.

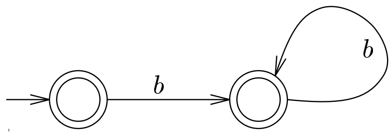


Figure 11: The *DFA* $Iconstr(b^*)$.

Example 5.70 (*Iconstr*): We construct the *DFA* corresponding to $(a \cup \epsilon)b^*$. The derivative item set is (the individual item sets have been compressed, as a notational convenience, and each item set is given a label):

$$\{I_0 = \bullet(\bullet(\bullet a \cup \bullet\epsilon\bullet)\bullet)(\bullet(\bullet b)^*\bullet)\bullet, I_2 = ((a\bullet \cup \epsilon)\bullet)(\bullet(\bullet b)^*\bullet)\bullet, I_1 = ((a \cup \epsilon))((\bullet b \bullet)^*\bullet)\bullet, I_3 = \emptyset\}$$

The *DFA* is shown in Figure 10 □

5.5.1 Making the construction more efficient

Because of the definition of $\mathcal{C}$, function *Iconstr* sometimes constructs a *DFA* which is larger than necessary, as shown in the following example.

Example 5.71 (A *DFA* that is not minimal): We use *Iconstr* to construct a *DFA* for $b^* \in RE$. The two item sets are $D = \{\{\bullet(b^*), (\bullet b)^*, (b^*)\bullet\}, \{(b\bullet)^*, (\bullet b)^*, (b^*)\bullet\}\}$. The *DFA* is shown in Figure 11. The problem is that the two item sets should have been recognized as denoting equivalent derivatives since:

$$\mathcal{L}_{IS}(\{\bullet(b^*), (\bullet b)^*, (b^*)\bullet\}) = \mathcal{L}_{IS}(\{(b\bullet)^*, (\bullet b)^*, (b^*)\bullet\})$$

They only differ in the items $\bullet(b^*)$ and $(b\bullet)^*$. □

The problem is that for some $J \in IS$, there is much redundant information in $\mathcal{C}(J)$. In particular, there may be a $J' \subset J$ such that $\mathcal{L}_{IS}(J') = \mathcal{L}_{IS}(J)$. We can introduce a function $\mathcal{X}$ such that $\mathcal{X} \circ \mathcal{C}$ is a closure function. That is, $\mathcal{X}$ is used as a filter.

Definition 5.72 (Item set optimization function $\mathcal{X}$): Given $J \in IS$ such that $J = \mathcal{C}(J)$, $\mathcal{X}(J)$ is the same as J, with the following removed: any item containing a subexpression of the form $\bullet(E \cup F)$, $\bullet(E^*)$, or $(E\bullet)^*$. □

Property 5.73 (Function $\mathcal{X} \circ \mathcal{C}$): Function $\mathcal{X} \circ \mathcal{C}$ satisfies Definition 5.62, and is a closure function. □

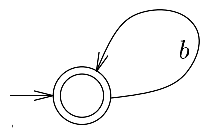


Figure 12: The *DFA* *DeRemer*(b^*).

Remark 5.74: The reason that such items are removed is that the definition of $\mathcal{C}$ ensures that they are redundant; other items will have been added to the item set to ensure that these ones are not needed. For example, in the case of $\bullet(E \cup F)$ function $\mathcal{C}$ will ensure that $\bullet E \cup F$ and $E \cup \bullet F$ are added to the item set. In this case $\mathcal{R}(\bullet(E \cup F)) = \mathcal{R}(\bullet E \cup F) \cup \mathcal{R}(E \cup \bullet F)$. □

Using composite function $\mathcal{X} \circ \mathcal{C}$, we can now present a revised version of the above construction.

Construction 5.75 (DeRemer's construction): DeRemer's construction is function *DeRemer*, which is exactly as *Iconstr*, except that the composite function $\mathcal{X} \circ \mathcal{C}$ is used as the closure function wherever $\mathcal{C}$ was used in *Iconstr*. This construction is due to DeRemer [DeRe74], where he attributes the idea behind the definition of $\mathcal{X}$ to Earley [Earl70]. □

Remark 5.76: DeRemer presented this construction in a slightly different context: he extended an LR parser to deal with grammar production rules with regular expression as right hand sides [DeRe74]. Remark 5.78 points out a slight problem with the original presentation by DeRemer. □

Example 5.77 (*DeRemer*): We use *DeRemer* to construct a *DFA* for $b^* \in RE$. The only item set is $\{(\bullet b)^*, (b^*)\bullet\}$ and the *DFA* is shown in Figure 12. With alphabet $V = \{b\}$, this is the minimal *Complete DFA* accepting $\mathcal{L}_{RE}(b^*)$. □

Remark 5.78: DeRemer and Earley specify that both the closure (function $\mathcal{C}$) and the optimization (function $\mathcal{X}$) operations are to be performed simultaneously. Unfortunately, when ϵ is permitted as an *RE* (as we have done) it is possible that the process never terminates. For example, consider the closure (with optimization) of $\{\bullet(a \cup \epsilon)^*\}$, in the style of DeRemer and Earley. After the first step we have $\{\bullet(a \cup \epsilon)^*, (\bullet(a \cup \epsilon))^*\}$. After an optimization step, and a few more steps we have $\{(\bullet a \cup \epsilon)^*, ((a \cup \epsilon)\bullet)^*\}$, after which we add $(\bullet(a \cup \epsilon))^*$ which we had originally removed. The rewriting process begins again. In this paper, we avoid this problem by defining the closure and optimization steps separately. □

We can devise an even more effective optimization function than $\mathcal{X}$.

Definition 5.79 (Function $\mathcal{Y}$): Given $J \in IS$ corresponding to $E \in RE$ such that $J = \mathcal{C}(J)$, $\mathcal{Y}(J)$ is a subset of J, keeping only the following items:

1. Any item containing a subexpression of the form $\bullet a$ (for some regular expression $a \in V$).
2. The item $E\bullet$ (if present in J).

□

Property 5.80 (Function $\mathcal{Y} \circ \mathcal{C}$): Composite function $\mathcal{Y} \circ \mathcal{C}$ satisfies Definition 5.62, and is a closure function. □

Remark 5.81: The function $\mathcal{Y}$ makes the computation of derivatives (of a $J \in IS$ such that $J = \mathcal{C}(J)$) particularly easy, as the items in $\mathcal{Y}(J)$ are precisely those required in the computation of derivatives and for determining if $\epsilon \in \mathcal{L}_{IS}(J)$. □

Construction 5.82 (Improved item sets): Our optimized construction, called *Oconstr*, is as *Iconstr*, except that the composite function $\mathcal{Y} \circ \mathcal{C}$ is used wherever $\mathcal{C}$ was used in *Iconstr*. This construction does not appear in the literature. □

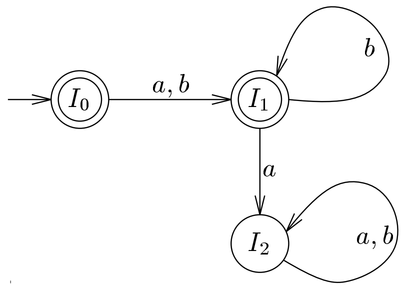
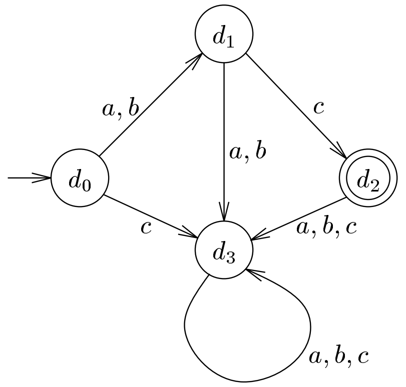


Figure 13: The *DFA* $Oconstr((a \cup \epsilon)b^*)$.



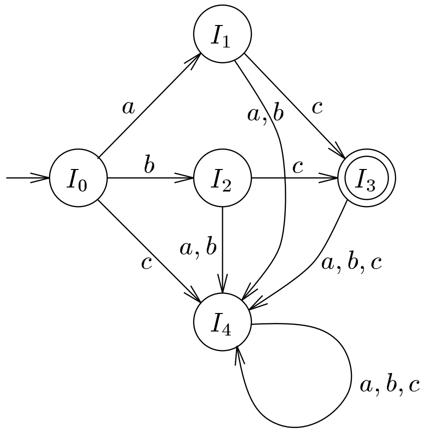


Figure 14: The *DFA*'s $Brz(ac \cup bc)$ and $Oconstr(ac \cup bc)$.

Example 5.83 (*Oconstr*): Using *Oconstr*, we construct a *DFA* for $(a \cup \epsilon)b^*$. The derivative item sets are (each item set is given a label for use in the state graph): $\{I_0 = \{(\bullet a \cup \epsilon)b^*, (a \cup \epsilon)(\bullet b)^*, (a \cup \epsilon)b^*\bullet\}, I_1 = \{(a \cup \epsilon)(\bullet b)^*, (a \cup \epsilon)b^*\bullet\}, I_2 = \emptyset\}$. The *DFA* is shown in Figure 13. This is the minimal *Complete DFA* for the given regular expression. □

As seen in the above example, function *Oconstr* constructs a smaller *DFA* than *Brz* did in Example 5.37. The two constructions seem difficult to compare, as the following example shows:

Example 5.84 (Comparing Brz to *Oconstr*): We use Brz and *Oconstr* to construct *DFA*'s for $ac \cup bc$. The derivatives (under extended similarity — see Remark 5.32, each given a label) are: $\{d_0 = ac \cup bc, d_1 = c, d_2 = \epsilon, d_3 = \emptyset\}$. The derivative item sets (using $\mathcal{Y} \circ \mathcal{C}$, each given a label) are: $\{I_0 = \{\bullet ac \cup bc, ac \cup \bullet bc\}, I_1 = \{a \bullet c \cup bc\}, I_2 = \{ac \cup b \bullet c\}, I_3 = \{(ac \cup bc)\bullet\}, I_4 = \emptyset\}$ The results are shown in Figure 14. Construction *Oconstr* is unable to recognize that states I_1 and I_2 are equivalent. (An equivalence relation on *IS* — much like $\sim$ on *ERE*'s — could be defined in order to identify such equivalent states.) □

We can make a more practical implementation by concatenating an end-marker \$ onto $E \in RE$ (using function $marker_e$ — see Definition 4.46). The second rule defining $\mathcal{Y}$ (Definition 5.79) is then no longer required.

Construction 5.85 (*Oconstr* with end-marker): Using the end-marker, the body of the *Oconstr* construction becomes (assuming that $DIS(E\$, D, \mathcal{Y} \circ \mathcal{C})$):

$$\begin{array}{ll}
\textbf{let} & T(J,a) = \{\mathcal{Y} \circ \mathcal{C}(a^{-1}J)\} \\
 & S = \{\mathcal{Y} \circ \mathcal{C}(\{\bullet E\})\} \\
 & F = \{J : J \in D \land \$^{-1}J \neq \emptyset\} \\
\textbf{in} & \\
 & (D, V, T, \emptyset, S, F) \\
\textbf{end} &
\end{array}$$

□

Remark 5.86: The Aho-Sethi-Ullman construction (Construction 4.50) can be viewed as a heavily encoded variation on the *Oconstr* construction. Each item in an item set of D is of the form $\ldots \bullet a \ldots$ (for $a \in V$) and corresponds to the basis *RFA*'s that are used in the construction of the *RFA* for E. The subset construction (with start-unreachable state removal) of the Aho-Sethi-Ullman algorithm (Algorithm 4.52) is folded into the algorithm computing the derivative item set (using composite function $\mathcal{Y} \circ \mathcal{C}$) and the definition of a derivative of an item set. Compare the *DFA* produced in Example 4.51 to that produced in Example 5.83; the only difference is the sink state in the latter example. □

6 Conclusions

The conclusions of this paper fall into two groups, depending on the section to which they relate: constructions based upon the structure of regular expressions (Section 4), or constructions based upon the Myhill-Nerode theorem (Section 5).

The conclusions about constructions based on regular expression structure are:

- Finite automaton constructions are frequently said to be "based upon the structure of regular expressions." The Σ-algebra framework (given in Sections 3 and 4) was useful in formalizing this notion. The Σ-algebras were particularly useful in the following ways:
  - They placed Thompson's, the left-biased, the right-biased, and the reduced finite automata (*RFA*) constructions in a common framework.
  - They highlighted the fact that the type of object produced by the Thompson's, the left-biased and the right-biased constructions is actually the isomorphism class of a finite automaton, as opposed to a finite automaton.
- The concept of *duality* (that one construction can be the mirror image of another) played a central part in finding common parts in constructions. Duality was made more obvious through the use of Σ-algebras. The following constructions were found to be related by duality:
  - The Berry-Sethi nondeterministic finite automaton construction (also known in the literature as the McNaughton-Yamada or the Glushkov nondeterministic finite automata construction) and the dual of the Berry-Sethi construction (a variant of which is also known as the Aho-Sethi-Ullman nondeterministic finite automata contruction [ASU86, Example 3.22, pg. 140]).
  - The McNaughton-Yamada-Glushkov deterministic finite automaton (*DFA*) construction and the Aho-Sethi-Ullman *DFA* construction[12].
- The use of end-markers (concatenated to either the left or the right of a regular expression) was found to be a simple coding trick, which may be useful in practice. End markers do not play a central role in any of the constructions, although they have previously been portrayed as important.
- The concept of marking a regular expression (each alphabet symbol occuring in the regular expression is given a unique mark, making all of the symbols unique — see Section 4.5) is an encoding trick. Marking is not central to the correctness of any of the constructions, although it is a useful technique in the practical implementation of some of the constructions.
- Marking was found to cause problems in some of the constructions. In particular, intersection, complementation, and language difference cannot be dealt with using marking[13]. In the Σ-algebra framework, intersection, complementation, and language difference can easily be implemented for the Berry-Sethi, McNaughton-Yamada-Glushkov, and Aho-Sethi-Ullman constructions — constructions that are all traditionally defined using marking.
- Two interpretations of marking appear in the literature. In the first one, being "at a mark" (Aho, Sethi, and Ullman use the phrase "at a position" [ASU86]) means to be in the state resulting from making a transition on the alphabet symbol associated with the particular mark[14]. The second interpretation equates being "at a mark" with being in the state which

[12] Here we assume that the sink state (if it exists) is removed from a *DFA* produced by the McNaughton-Yamada-Glushkov construction.

[13] Actually, McNaughton and Yamada [MY60] attempted to define intersection and complementation. Their informal descriptions are difficult to understand, and more recent papers use marking and have abandoned trying to define intersection or complementation. See Section 4.4.

[14] This is the interpretation taken by Glushkov, McNaughton and Yamada, and Berry and Sethi [Glus61, MY60, BS86].

has an out-transition on the alphabet symbol associated with the particular mark (that is, the marked symbol is valid as the next input symbol)[15]. The two interpretations are duals of one another, and arise naturally from the duality of the left-biased and right-biased constructions. For example, the interpretations give rise to the duality between McNaughton, Yamada and Glushkov's *DFA* and Aho, Sethi, and Ullman's *DFA* constructions.

- The improvements to the Berry-Sethi construction[16] due to Brüggemann-Klein [B-K93a] and Chang and Paige [Chan92, CP92] have been difficult to compare. This has been largely due to the fact that Chang and Paige's improvements are to the finite automaton construction itself, while Brüggemann-Klein's improvements involve transforming the regular expression. In Section 4.5, we presented an improvement to the construction (not found in the literature) that mirrors Brüggemann-Klein's improvements (not on regular expressions, but on finite automata), and is easy to compare to Chang and Paige's construction.

- Some relationships between the constructions were found that were not made obvious by the Σ-algebra derivations:

  - For restricted regular expressions (where each alphabet symbol occurs at most once — as in marked regular expressions) the Berry-Sethi construction produces a deterministic *FA*. As a consequence, the Berry-Sethi construction and the McNaughton-Yamada-Glushkov *DFA* construction produce isomorphic finite automata (with the exception of the sink state present in a McNaughton-Yamada-Glushkov *DFA*).
  - The Berry-Sethi construction (and therefore the McNaughton-Yamada-Glushkov *DFA* construction) and the Brzozowski construction (under an appropriate encoding) produce isomorphic finite automata for restricted regular expressions. This result was originally presented by Berry and Sethi [BS86].

The conclusions about the Myhill-Nerode, Brzozowski, and DeRemer constructions (Section 5) are:

- Deriving the second major family of constructions from the Myhill-Nerode theorem proved useful in a number of ways:

  - The use of equivalence classes makes the correctness argument for the Myhill-Nerode construction particularly clear.
  - The unique minimal *DFA* (for a particular language) can be easily constructed using a particular equivalence class as the parameter to the Myhill-Nerode construction.
  - Derivatives (of a language) are a useful encoding of the equivalence classes of Myhill and Nerode's unique minimal-index equivalence relation R_L.
  - The definition of derivatives provides an efficient method to compute finite sets which encode the infinite sets that are used in the Myhill-Nerode construction.

- The Brzozowski construction can be viewed as an ingenious encoding of the Myhill-Nerode minimal *DFA* construction.

- Brzozowski's original paper provided a proof (a similar one is given in this paper) that his construction also works when only similarity of regular expressions is recognized. Similarity is defined in his paper using four rules, and is defined in this paper using only three rules. The missing fourth rule (that ϵ is the unit of concatenation) is not required in the definition of similarity for the correctness of our presentation of Brzozowski's construction.

[15] This is the interpretation taken by Aho, Sethi, and Ullman [ASU86].

[16] Which is therefore an improvement to the McNaughton-Yamada-Glushkov and the Aho-Sethi-Ullman nondeterministic finite automaton constructions.

- We defined the equivalence and the similarity of regular expressions as equivalence relations on regular expressions. We also demonstrated that any such equivalence relation E can be used in Brzozowski's construction, provided that E is a refinement of equivalence and similarity is a refinement of E.
- For restricted regular expressions, Brzozowski's construction (with an encoding), the Berry-Sethi construction, and the McNaughton-Yamada-Glushkov construction produce isomorphic *DFA*'s.
- The use of dotted regular expressions (also known as *items,* from LR parsing) is a useful, if obscure, encoding of the derivatives of a regular expression. We obtained the following results on the use of dotted regular expressions:
  - Computing the set of all dotted regular expressions (from a given regular expression) can be defined very simply. The derivatives of dotted regular expressions, and the construction of a *DFA* can be defined simply. This construction does not appear in the literature.
  - The straightforward definition of dotted regular expressions is unable to deal with intersection and complementation. This is for the same reason that marking constructions are unable to deal with intersection and complementation.
  - DeRemer specified a *DFA* construction that appears to be very easy to implement. It is an optimization over the straightforward dotted regular expression construction, and the constructed *DFA* is always smaller.
  - The original specification of item closure, due to DeRemer and Earley, is incomplete. They attempted to define closure and optimization as a single step. This can lead to non-termination, as we have demonstrated. The problem can be easily solved by defining closure and optimization steps separately.
  - We show that additional optimizations, added to DeRemer's construction, can reduce the size of the produced *DFA*. This construction is not given in the literature. Furthermore, the optimizations are arguably easier to understand than those of DeRemer, and likely easier to implement.
  - It is possible to show that this improved construction is related to the Aho-Sethi-Ullman *DFA* construction.

A Some basic definitions

Convention A.1 (Powerset): For any set A we use $\mathcal{P}(A)$ to denote the set of all subsets of A. $\mathcal{P}(A)$ is called the *powerset* of A; it is sometimes written 2^A. $\Box$

Convention A.2 (Sets of functions): For sets A and B, $A \longrightarrow B$ denotes the set of all total functions from A to B, while $A \not\longrightarrow B$ denotes the set of all partial functions from A to B. $\Box$

Remark A.3: For sets A, B and relation $C \subseteq A \times B$ we can interpret C as a function $C \in A \longrightarrow \mathcal{P}(B)$. $\Box$

Convention A.4 (Tuple projection): For an n-tuple $t = (x_1, x_2, \ldots, x_n)$ we use the notation $\pi_i(t)$ $(1 \leq i \leq n)$ to denote tuple element x_i; we use the notation $\bar{\pi}_i(t)$ $(1 \leq i \leq n)$ to denote the $(n-1)$-tuple $(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots x_n)$. Both π and $\bar{\pi}$ extend naturally to sets of tuples. $\Box$

Convention A.5 (Tuple arguments to functions): For functions (or predicates) taking a single tuple as an argument, we usually drop one set of parentheses in a function application. $\Box$

Convention A.6 (Relation composition): Given sets A, B, C (not necessarily different) and two relations, $E \subseteq A \times B$ and $F \subseteq B \times C$, we define relation composition (infix operator $\circ$) as:

$$E \circ F = \{(a, c) : (\exists\ b : b \in B : (a, b) \in E \wedge (b, c) \in F)\}$$

$\Box$

Convention A.7 (Equivalence classes of an equivalence relation): For any equivalence relation E on set A we denote the set of equivalence classes of E by $[A]_E$; that is

$$[A]_E = \{[a]_E : a \in A\}$$

Set $[A]_E$ is also called the *partition* of A induced by E. $\Box$

Definition A.8 (Index of an equivalence class): For equivalence relation E on set A, define $\sharp E = |[A]_E|$. $\sharp E$ is called the *index* of E. $\Box$

Definition A.9 (Alphabet): An *alphabet* is a non-empty set of finite size. $\Box$

Definition A.10 (Refinement of an equivalence relation): For equivalence relations E and E' (on set A), E is a *refinement* of E' if and only if $E \subseteq E'$. $\Box$

Definition A.11 (Refinement ($\sqsubseteq$) relation on partitions): For equivalence relations E and E' (on set A), $[A]_E$ is said to be a refinement of $[A]_{E'}$ (written $[A]_E \sqsubseteq [A]_{E'}$) if and only if $E \subseteq E'$. An equivalent statement is that $[A]_E \sqsubseteq [A]_{E'}$ if and only if every equivalence class (of A) under E is entirely contained in some equivalence class (of A) under E'. $\Box$

Property A.12 (Equivalence relations): Given two equivalence relations E, F, we have the following property:

$$(E \subseteq F) \wedge (\sharp E = \sharp F) \Rightarrow (E = F)$$

$\Box$

Definition A.13 (Regular languages): $\mathcal{L}_{\text{reg}_V}$ denotes the set of all regular languages over alphabet V. That is, $\mathcal{L}_{\text{reg}_V} \subseteq \mathcal{P}(V^*)$ is the smallest set containing V that is closed under $\cup$ (language union), $\cdot$ (a dot, language concatenation), and $*$ (Kleene closure). The subscript V is dropped when no ambiguity arises. $\Box$

Definition A.14 (Operator ? on languages): We define ? as a postfix (superscript) operator on languages as $L^? = L \cup \{\epsilon\}$. $\Box$

Definition A.15 (Left derivatives): Given language $A \subseteq V^*$ and $w \in V^*$ we define the *left derivative* of A with respect to w as:

$$w^{-1}A = \{x \in V^* : wx \in A\}$$

Sometimes derivatives are written as $D_w A$ or as $\frac{dA}{dw}$. Right derivatives are analogously defined. Derivatives can also be extended to $B^{-1}A$ where B is also a language. □

Property A.16 (Left derivatives): The following two properties follow from Definition A.15 (assuming L is a language):

- $w \in L \equiv \epsilon \in w^{-1}L$, and
- $(wa)^{-1}L = a^{-1}(w^{-1}L)$.

□

Property A.17 (Derivatives of regular languages): Assuming $a \in V$ and $L, L_0, L_1 \in \mathcal{L}_{\text{reg}}$, derivatives have the following properties (given with respect to the structure of regular languages):

$$\begin{array}{rcl}
a^{-1}\emptyset & = & \emptyset \\
a^{-1}\{\epsilon\} & = & \emptyset \\
a^{-1}\{b\} & = & \textbf{if } (a = b) \textbf{ then } \{\epsilon\} \textbf{ else } \emptyset \textbf{ fi} \\
a^{-1}(L_0 L_1) & = & (a^{-1}L_0)L_1 \cup \textbf{if } (\epsilon \in L_0) \textbf{ then } a^{-1}L_1 \textbf{ else } \emptyset \textbf{ fi} \\
a^{-1}(L_0 \cup L_1) & = & (a^{-1}L_0) \cup (a^{-1}L_1) \\
a^{-1}(L^*) & = & (a^{-1}L)L^* \\
a^{-1}(L^+) & = & (a^{-1}L)L^* \\
a^{-1}(L^?) & = & a^{-1}L \\
a^{-1}(L_0 \cap L_1) & = & (a^{-1}L_0) \cap (a^{-1}L_1) \\
a^{-1}(\neg L) & = & \neg(a^{-1}L)
\end{array}$$

The definition related to Kleene closure is shown as follows:

$$\begin{array}{ll}
 & a^{-1}(L^*) \\
= & \quad \{\text{Definition of } * \} \\
 & a^{-1}((L \setminus \{\epsilon\})L^* \cup \{\epsilon\}) \\
= & \quad \{\text{Definition of } a^{-1}(L_0 \cup L_1) \} \\
 & a^{-1}((L \setminus \{\epsilon\})L^*) \cup a^{-1}\{\epsilon\} \\
= & \quad \{\text{Definition of derivative of concatenation and } \{\epsilon\} \} \\
 & (a^{-1}(L \setminus \{\epsilon\}))L^* \\
= & \quad \{\text{Definition of derivative of } \{\epsilon\} \} \\
 & (a^{-1}L)L^*
\end{array}$$

The definition related to complementation $\neg$ is as follows:

$$\begin{array}{ll}
 & a^{-1}(\neg L) \\
= & \quad \{\text{Definition of derivative} \} \\
 & \{x : ax \in \neg L\} \\
= & \quad \{\text{Definition of } \neg \text{ operator} \} \\
 & \neg\{x : ax \in L\} \\
= & \quad \{\text{Definition of } a^{-1}L \} \\
 & \neg(a^{-1}L)
\end{array}$$

□

Definition A.18 (Preserving a predicate): A (partial) function $f \in B^n \not\longrightarrow B$ (for fixed $n \geq 0$) is said to *preserve* predicate (or property) P (on B) if and only if

$$(\forall\ B' : B' \in B^n \cap (domain(f)) \wedge (\forall\ k : 1 \leq k \leq n : P(\pi_k(B'))) : P(f(B')))$$

The set $domain(f)$ refers to those elements of B^n on which f is defined. □

Intuitively, a function f preserves a property P if, when every argument of f satisfies P, the result of f applied to the arguments also satisfies P.

Definition A.19 (Reversal operator): A reversal operator R (usually written postfix and superscript) for a set A is a function $R \in A \longrightarrow A$ such that $R \circ R$ (equivalently R^2) is the identity function on A. We sometimes write the reversal operator as a standard (prefix notation) function. □

Definition A.20 (Tuple and relation reversal): For an n-tuple $(x_1, x_2, \ldots, x_n)$ define reversal as (postfix and superscript) function R:

$$(x_1, x_2, \ldots, x_n)^R = (x_n, \ldots, x_2, x_1)$$

Given a set A of tuples, we define $A^R = \{x^R : x \in A\}$. □

Definition A.21 (Dual of a function): We assume two sets A and B whose reversal operators are R and R' respectively. Two functions, $f \in A \longrightarrow B$ and $f_d \in A \longrightarrow B$ are one another's *dual* if and only if

$$f(a) = (f_d(a^R))^{R'}$$

In some cases we relax the equality to isomorphism (when isomorphism is defined on B). □

Definition A.22 (Symmetrical function): A *symmetrical* function is one that is its own dual. □

Proposition A.23 (Symmetrical functions): The composition of two symmetrical functions is again symmetrical. □

B Proofs of some Σ-algebra operators

In this section, we sketch proofs of the correctness of the operators of Thompson's Σ-algebra (Definition 4.1), the left-biased Σ-algebra operators (Definition 4.20), the Chang-Paige *RFA'* operators (Construction 4.77), and the *RFA''* operators (Construction 4.79).

Theorem B.1 (Correctness of Thompson's Σ-algebra of *FA*'s): Recall the operator definitions of Definition 4.1. We only present a correctness proof for the operator $C_{\cup,Th}$. In the following derivation we assume the context of the innermost **let** clause of the operator definition.

$$\begin{array}{ll}
& \mathcal{L}_{FA}(C_{\cup,Th}([M_0]_{\cong},[M_1]_{\cong})) \\
= & \quad \{\text{Definition of } \mathcal{L}_{FA}\} \\
& (T_0 \cup T_1)^*(q_0,q_1) \\
= & \quad \{\text{Definition of } E'\} \\
& (\cup\ s,f : s \in S_0 \wedge f \in F_0 : T_0^*(s,f)) \cup (\cup\ s,f : s \in S_1 \wedge f \in F_1 : T_1^*(s,f)) \\
= & \quad \{\text{Definitions of } \mathcal{L}_{FA}([M_0]_{\cong}), \mathcal{L}_{FA}([M_1]_{\cong})\} \\
& \mathcal{L}_{FA}([M_0]_{\cong}) \cup \mathcal{L}_{FA}([M_1]_{\cong})
\end{array}$$

□

Theorem B.2 (Correctness of the *LBFA* operators): Recall the operator definitions of Definition 4.20. We present a correctness proof of the operator $C_{\cup,LBFA}$. In the following derivation we assume the context of the innermost **let** clause of the operator definition.

$$\begin{array}{ll}
& \mathcal{L}_{FA}(C_{\cup,LBFA}([M_0]_{\cong},[M_1]_{\cong})) \\
= & \quad \{\text{Definitions of } \mathcal{L}_{FA} \text{ and } F'\} \\
& (\cup\ f : f \in F_0 \cap Q' : T'^*(q_0,f)) \cup (\cup\ f : f \in F_1 \cap Q' : T'^*(q_0,f)) \\
& \quad \cup\, \mathbf{if}\ (N)\ \mathbf{then}\ T'^*(q_0,q_0)\ \mathbf{else}\ \emptyset\ \mathbf{fi} \\
= & \quad \{\text{Definitions of } \mathcal{L}_{FA}([M_0]_{\cong}), \mathcal{L}_{FA}([M_1]_{\cong}), N, T'\} \\
& \mathcal{L}_{FA}([M_0]_{\cong}) \cup \mathcal{L}_{FA}([M_1]_{\cong}) \cup \mathbf{if}\ (\epsilon \in (\mathcal{L}_{FA}(M_0) \cup \mathcal{L}_{FA}(M_1)))\ \mathbf{then}\ \{\epsilon\}\ \mathbf{else}\ \emptyset\ \mathbf{fi} \\
= & \quad \{\text{Definition of } \mathcal{L}_{FA}([M_0]_{\cong}) \cup \mathcal{L}_{FA}([M_1]_{\cong})\} \\
& \mathcal{L}_{FA}([M_0]_{\cong}) \cup \mathcal{L}_{FA}([M_1]_{\cong})
\end{array}$$

□

Theorem B.3 (Correctness of the Chang-Paige *RFA'* operators): Recall Definition 4.76 and Construction 4.77. We only present the derivation of the eighth component (usually called W) for operators $C_{\cdot,RFA'}$ and $C_{\cup,RFA'}$ (the others are easy to prove). We assume the context of the innermost **let** clause for both operators.

$$\begin{array}{lll}
C_{\cdot,RFA'}\textbf{:} & & \mathit{last}' \times \mathit{first}' \setminus (\mathit{follow}_0 \uplus \mathit{follow}_1 \uplus \mathit{last}_0 \times \mathit{first}_1) \\
& = & \quad \{\text{Definitions of } \mathit{first}' \text{ and } \mathit{last}'\} \\
& & (\mathit{last}_1 \uplus \mathbf{if}\ (\mathit{null}_1)\ \mathbf{then}\ \mathit{last}_0\ \mathbf{else}\ \emptyset\ \mathbf{fi}) \times (\mathit{first}_0 \uplus \mathbf{if}\ (\mathit{null}_0)\ \mathbf{then}\ \mathit{first}_1\ \mathbf{else}\ \emptyset\ \mathbf{fi}) \\
& & \quad \setminus (\mathit{follow}_0 \uplus \mathit{follow}_1 \uplus \mathit{last}_0 \times \mathit{first}_1) \\
& = & \quad \{\text{Rewriting}\} \\
& & (\mathit{last}_1 \times \mathit{first}_0 \uplus \mathbf{if}\ (\mathit{null}_0)\ \mathbf{then}\ \mathit{last}_1 \times \mathit{first}1\ \mathbf{else}\ \emptyset\ \mathbf{fi} \\
& & \quad \uplus\, \mathbf{if}\ (\mathit{null}_1)\ \mathbf{then}\ \mathit{last}_0 \times \mathit{first}_0\ \mathbf{else}\ \emptyset\ \mathbf{fi} \\
& & \quad \uplus\, \mathbf{if}\ (\mathit{null}_0 \wedge \mathit{null}_1)\ \mathbf{then}\ \mathit{last}_0 \times \mathit{first}_1\ \mathbf{else}\ \emptyset\ \mathbf{fi} \\
& & \quad \setminus (\mathit{follow}_0 \uplus \mathit{follow}_1 \uplus \mathit{last}_0 \times \mathit{first}_1) \\
& = & \quad \{\text{Assumption that } Q_0 \cap Q_1 = \emptyset\} \\
& & \mathit{last}_1 \times \mathit{first}_0 \uplus \mathbf{if}\ (\mathit{null}_0)\ \mathbf{then}\ \mathit{last}_1 \times \mathit{first}_1 \setminus \mathit{follow}_1\ \mathbf{else}\ \emptyset\ \mathbf{fi}
\end{array}$$

$$
\begin{array}{ll}
& \quad \uplus \textbf{ if } (null_1) \textbf{ then } last_0 \times first_0 \setminus follow_0 \textbf{ else } \emptyset \textbf{ fi} \\
= & \qquad \{ \text{Definitions of } W_0 \text{ and } W_1 \} \\
& last_1 \times first_0 \uplus \textbf{if } (null_0) \textbf{ then } W_1 \textbf{ else } \emptyset \textbf{ fi} \uplus \textbf{if } (null_1) \textbf{ then } W_0 \textbf{ else } \emptyset \textbf{ fi} \\
= & \qquad \{ \text{Definitions of } W' \text{ and } W'' \} \\
& last_1 \times first_0 \uplus W' \uplus W''
\end{array}
$$

$$
\begin{array}{ll}
C_{\cup,RFA'}\textbf{:} & (last_0 \uplus last_1) \times (first_0 \uplus first_1) \setminus (follow_0 \uplus follow_1) \\
= & \qquad \{ \text{Rewriting} \} \\
& (last_0 \times first_0 \uplus last_0 \times first_1 \uplus last_1 \times first_0 \uplus last_1 \times first_1) \setminus (follow_0 \uplus follow_1) \\
= & \qquad \{ \text{Assumption that } Q_0 \cap Q_1 = \emptyset \} \\
& (last_0 \times first_0 \setminus follow_0) \uplus last_0 \times first_1 \uplus last_1 \times first_0 \uplus (last_1 \times first_1 \setminus follow_1) \\
= & \qquad \{ \text{Definitions of } W_0 \text{ and } W_1 \} \\
& last_0 \times first_1 \uplus last_1 \times first_0 \uplus W_0 \uplus W_1
\end{array}
$$

□

Theorem B.4 (Correctness of the RFA'' operators): Recall Definition 4.78 and Construction 4.79. We only present the derivation of the eighth component (usually called W) for operators $C_{\cdot,RFA''}$ and $C_{\cup,RFA''}$ (the others are easy to prove). We assume the context of the innermost **let** clause for both operators.

$$
\begin{array}{ll}
C_{\cdot,RFA''}\textbf{:} & (follow_0 \uplus follow_1 \uplus (last_0 \times first_1)) \setminus (last' \times first') \\
= & \qquad \{ \text{Definitions of } first' \text{ and } last' \} \\
& (follow_0 \uplus follow_1 \uplus (last_0 \times first_1)) \setminus ((last_1 \uplus \textbf{if } (null_1) \textbf{ then } last_0 \textbf{ else } \emptyset \textbf{ fi}) \\
& \quad \times (first_0 \uplus \textbf{if } (null_0) \textbf{ then } first_1 \textbf{ else } \emptyset \textbf{ fi})) \\
= & \qquad \{ \text{Rewriting} \} \\
& (follow_0 \uplus follow_1 \uplus (last_0 \times first_1)) \setminus (last_1 \times first_0 \\
& \quad \uplus \textbf{ if } (null_0) \textbf{ then } last_1 \times first_1 \textbf{ else } \emptyset \textbf{ fi} \\
& \quad \uplus \textbf{ if } (null_1) \textbf{ then } last_0 \times first_0 \textbf{ else } \emptyset \textbf{ fi} \\
& \quad \uplus \textbf{ if } (null_0 \wedge null_1) \textbf{ then } last_0 \times first_1 \textbf{ else } \emptyset \textbf{ fi}) \\
= & \qquad \{ \text{Assumption that } Q_0 \cap Q_1 = \emptyset \} \\
& \textbf{if } (null_0) \textbf{ then } follow_1 \setminus last_1 \times first_1 \textbf{ else } follow_1 \textbf{ fi} \\
& \quad \uplus \textbf{ if } (null_1) \textbf{ then } follow_0 \setminus last_0 \times first_0 \textbf{ else } follow_0 \textbf{ fi} \\
& \quad \uplus \textbf{ if } (null_0 \wedge null_1) \textbf{ then } last_0 \times first_1 \setminus last_0 \times first_1 \textbf{ else } last_0 \times first_1 \textbf{ fi} \\
= & \qquad \{ \text{Definitions of } W_0 \text{ and } W_1\text{; rewriting} \} \\
& \textbf{if } (null_0) \textbf{ then } W_1 \textbf{ else } follow_1 \textbf{ fi} \uplus \textbf{if } (null_1) \textbf{ then } W_0 \textbf{ else } follow_0 \textbf{ fi} \\
& \quad \uplus \textbf{ if } (null_0 \wedge null_1) \textbf{ then } \emptyset \textbf{ else } last_0 \times first_1 \textbf{ fi} \\
= & \qquad \{ \text{Definitions of } W', W'', \text{ and } W''' \} \\
& W' \uplus W'' \uplus W'''
\end{array}
$$

$$
\begin{array}{ll}
C_{\cup,RFA''}\textbf{:} & (follow_0 \uplus follow_1) \setminus ((last_0 \uplus last_1) \times (first_0 \uplus first_1)) \\
= & \qquad \{ \text{Rewriting} \} \\
& (follow_0 \uplus follow_1) \setminus (last_0 \times first_0 \uplus last_0 \times first_1 \uplus last_1 \times first_0 \uplus last_1 \times first_1) \\
= & \qquad \{ \text{Assumption that } Q_0 \cap Q_1 = \emptyset \} \\
& follow_0 \setminus (last_0 \times first_0) \uplus follow_1 \setminus (last_1 \times first_1) \\
= & \qquad \{ \text{Definitions of } W_0 \text{ and } W_1 \} \\
& W_0 \uplus W_1
\end{array}
$$

□

References

[ASU86] AHO, A.V., R. SETHI, AND J.D. ULLMAN. *Compilers: Principles, Techniques, and Tools,* Addison-Wesley Publishing Co., Reading, M.A., 1988.

[AU92] AHO, A.V. AND J.D. ULLMAN. *Foundations of Computer Science,* Computer Science Press, New York, N.Y. 1992.

[BL77] BACKHOUSE, R.C. AND R.K. LUTZ. "Factor graphs, failure functions and bi-trees," in *Fourth Colloquium on Automata, Languages and Programming,* (G. Goos and J. Hartmanis, eds.), pp. 61–75, Lecture Notes in Computer Science 52, Springer-Verlag, Berlin, 1977.

[BS86] BERRY, G. AND R. SETHI. "From regular expressions to deterministic automata," Theoretical Computer Science, 48: 117–126, 1986.

[B-K93a] BRÜGGEMANN-KLEIN, A. "Regular expressions into finite automata," to appear in Theoretical Computer Science.

[B-K93b] BRÜGGEMANN-KLEIN, A. *Private communication,* July 1993.

[Brzo64] BRZOZOWSKI, J.A. "Derivatives of regular expressions," J. ACM 11(4): 481–494, 1964.

[Cham93] CHAMPARNAUD, J.M. "From a regular expression to an automaton," Technical report, IBP, LITP, Université Paris 7, Paris, France, Working document 23 September 1993.

[Chan92] CHANG, C.-H. "From regular expressions to DFAs using compressed NFAs," PhD thesis, Computer Science Department, Courant Institute of Mathematical Sciences, New York University, N.Y., Oct. 1992.

[CP92] CHANG, C.-H. AND R. PAIGE. "From regular expressions to DFAs using compressed NFAs," Computer Science Department, Courant Institute of Mathematical Sciences, New York University, N.Y., 1992.

[DeRe74] DEREMER, F.L. "Lexical analysis," in *Compiler Construction: an Advanced Course,* (F.L. Bauer and J. Eickel, eds.), pp. 109–120, Lecture Notes in Computer Science 21, Springer-Verlag, Berlin, 1974.

[Dijk76] DIJKSTRA, E.W. *A discipline of programming,* Prentice-Hall Inc., N.J., 1976.

[Earl70] EARLEY, J. "An efficient context-free parsing algorithm," C. ACM 13(2): 94–102, Feb. 1970.

[EM85] EHRIG, E. AND B. MAHR. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics,* Springer, Berlin, 1985.

[tEvG93] TEN EIKELDER, H.M.M. AND H.P.J. VAN GELDROP. "On the correctness of some algorithms to generate finite automata for regular expressions," Computing Science Note 93/32, Eindhoven University of Technology, The Netherlands, 1993.

[Glus61] GLUSHKOV, V.M. "The abstract theory of automata," Russian Mathematical Surveys 16: 1–53, 1961.

[GJ90] GRUNE, D. AND C.J.H. JACOBS. *Parsing Techniques: A Practical Guide,* Ellis Horwood Ltd., West Sussex, England, 1990.

[HU79] HOPCROFT, J.E. AND J.D. ULLMAN. *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley Publishing Co., Reading, M.A., 1979.

[Knut65] KNUTH, D.E. "On the translation of languages from left to right," Inform. Control 8: 607–639, 1965.

[MY60] MCNAUGHTON, R. AND H. YAMADA. "Regular expressions and state graphs for automata," IEEE Trans. on Electronic Computers 9(1): 39–47, 1960.

[Myhi57] MYHILL, J. "Finite automata and the representation of events," WADD TR-57-624, pp. 112–137, Wright Patterson AFB, Ohio, 1957.

[Nero58] NERODE, A. "Linear automaton transformations," Proc. AMS 9: 541–544, 1958.

[RS59] RABIN, M.O AND D. SCOTT. "Finite automata and their decision problems," IBM J. Res. 3(2): 115–125, 1959.

[SS-S88] SIPPU, S. AND E. SOISALON-SOININEN. *Parsing Theory: Languages and Parsing,* Vol. 1, Springer-Verlag, Berlin, 1988.

[Thom68] THOMPSON, K. "Regular expression search algorithms," C. ACM 11(6): 419–422, 1968.

[Wats93] WATSON, B.W. "A taxonomy of finite automata minimization algorithms," Computing Science Note 93/44, Eindhoven University of Technology, The Netherlands, 1993.

[Wood87] WOOD, D. *Theory of Computation,* Harper & Row, Publishers, New York, N.Y., 1987.

Index