ECS 120 Lesson 8 – Regular Expressions, Pt. 2

Oliver Kreylos

Monday, April 16th, 2001

1 Equivalence of Regular Expressions and Finite State Machines, Contd.

Today we will prove the other direction of the proof started last time: The language accepted by any NFA M can also be generated by a regular expression R. We will prove this by giving an algorithm to convert an NFA into an equivalent regular expression. Before we can do this, we have to introduce yet another machine type, a *generalized nondeterministic finite automaton (GNFA)*.

2 Generalized Nondeterministic Finite Automata

A GNFA is very similar to a usual NFA; the main difference is, that each transition arrow is not labeled with a single character (or the symbol ϵ), but with a complete regular expression. For convenience, we will denote the set of all possible regular expressions over the alphabet Σ as $\mathcal{R}(\Sigma)$. We also add some minor requirements to make the following proof a little easier: We require that there are no transition arrows going into the start state; that there is exactly one final state; that there are no transition arrows leaving from the final state; and that there is a transition arrow going from each state to every other state (including itself).

2.1 Formal Definition of GNFAs

A GNFA M is a five-tuple $M = (Q, \Sigma, \delta, q_{\text{start}}, q_{\text{final}})$, where

- Q is a finite set of states,
- Σ is the input alphabet,
- $\delta\colon \big(Q \setminus \{q_{\text{final}}\}\big) \times \big(Q \setminus \{q_{\text{start}}\}\big) \to \mathcal{R}(\Sigma)$ is the transition function,
- q_{start} is the start state, and
- q_{final} is the final state.

Note the different way the transition function is defined: For a DFA, the transition function takes a state and an input character as arguments and returns the next state in the computation; for a GNFA, the transition function takes two states $q_1 \neq q_{\text{final}}$ and $q_2 \neq q_{\text{start}}$ as arguments and returns the regular expression written along the transition arrow from q_1 to q_2. This is a valid definition, since we required that there is a transition arrow from each state (except the final state) to each other state (except the start state).

2.2 Computation of GNFAs

As opposed to DFAs or NFAs, which read a single input character when transitioning from one state to the next, a GNFA transitions from one state to the next by reading a block of input characters that matches the regular expression the transition arrow is labeled with. A GNFA accepts a word $w \in \Sigma^*$, iff w can be written as a concatenation $w = w_1 w_2 \dots w_n$ for some $n \geq 0$ and words $w_1, w_2, \dots, w_n \in \Sigma^*$, such that there exists a sequence of states $(q_0, q_1, \dots, q_n) \in Q^n$ with the conditions

1. $q_0 = q_{\text{start}}$ is the start state,
2. $q_n = q_{\text{final}}$ is the final state, and
3. For all $i \in \{1, 2, \dots, n\} : w_i \in L\big(\delta(q_{i-1}, q_i)\big)$. In other words, each w_i must match the regular expression written along the transition arrow from q_{i-1} to q_i.

For DFAs and NFAs, we previously defined their computation more formally by constructing their extended transition functions δ^*. For GNFAs, we can proceed in a similar fashion. The main difference is, that $\delta^*(q, w)$ for a DFA/NFA is the set of states reachable from q when reading the word w; for GNFAs, the extended transition function $\delta^*(q_1, q_2)$ will give the regular expression that defines the words that are spelled out by all possible ways to reach state q_2 from state q_1. Finally, $\delta^*(q_{\text{start}}, q_{\text{final}})$ will be the regular expression defining all words that are spelled out by any chain of states from q_{start} to q_{final}, i. e., it will give the regular expression defining exactly the language of the GNFA. If we can find some algorithm to compute δ^*, we therefore have found an algorithm to compute the regular expression equivalent to a GNFA.

2.3 Equivalence of GNFAs and NFAs

Before we can proceed with the proof, we have to show that GNFAs do not introduce a new class of languages; in other words, that they are equivalent to NFAs. The first direction is easy: Starting from any NFA, we can add a new start state and a new final state without changing its language. Then we label each transition arrow with the regular expression defining its old label (either a single character or the symbol ϵ). Finally, we include all missing transition arrows and label them with the regular expression $\emptyset$. These changes do not change the language accepted by the NFA, so it is equivalent to the GNFA.

To prove the other direction, we observe that we already know how to construct an NFA that accepts the language generated by a regular expression. For each transition arrow in the GNFA, we insert the complete automaton accepting the language generated by the transition arrow's label as a "subautomaton;" this way, we can replace each regular expression by a set of states and character transitions, see Figure 1. In the end, we will have constructed an NFA equivalent to the GNFA.

3 The Conversion Algorithm

To convert an arbitrary NFA M with n states to an equivalent regular expression, we follow the following steps (see Figure 2):

1. Convert M to the equivalent GNFA M_{n+2} by adding a new start and final state. The new machine will have $n + 2$ states.

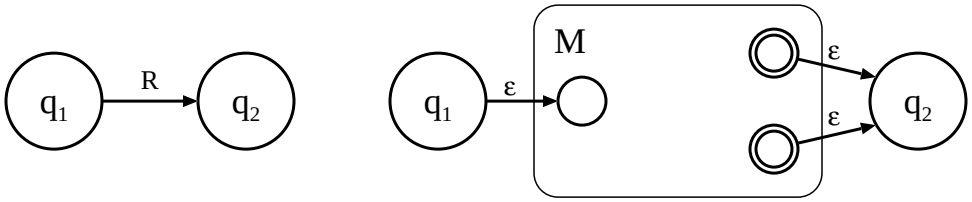

Figure 1: Replacing a transition labeled with a regular expression R in a GNFA by an NFA M accepting $L(R)$.

2. Say the current machine is M_k, having k states. Pick any state $s \in Q_k \setminus \{q_{\text{start}}, q_{\text{final}}\}$.

3. Eliminate s according to the following algorithm, yielding an equivalent machine M_{k-1} having one less state.

4. Repeat from step 2 while $k > 2$.

5. The last generated machine, M_2, will have two states – one is the start state, one is the final state. There will be exactly one arrow going from the start to the final state, labeled with a regular expression R. According to the definition of computation of a GNFA, all words accepted by M_2 are generated by R and vice versa. Therefore, R is a regular expression equivalent to M_2, and therefore by transitivity equivalent to M.

3.1 The Elimination Step

To eliminate a state $s \in Q \setminus \{q_{\text{start}}, q_{\text{final}}\}$ and still get a machine that accepts the same language, we have to "repair" the transition function. To do so, we have to understand how removing a state from the machine changes the way its computation works. Let $q_i, q_j \in Q \setminus \{s\}$ be two states, where $q_i \neq q_{\text{final}}$ and $q_j \neq q_{\text{start}}$. To go from state q_i to state q_j, the original machine could have taken the direct transition, labeled with a regular expression $R_4 := \delta(q_i, q_j)$. On the other hand, it could also have taken the route via s, first going from q_i to s, then going from s back to s any number of times, and finally going from s to q_j. Along the way, the machine would have read characters matching the regular expressions $R_1 := \delta(q_i, s)$, $R_2 := \delta(s, s)$ (any number of times) and $R_3 := \delta(s, q_j)$, see Figure 3. The machine could also have taken other

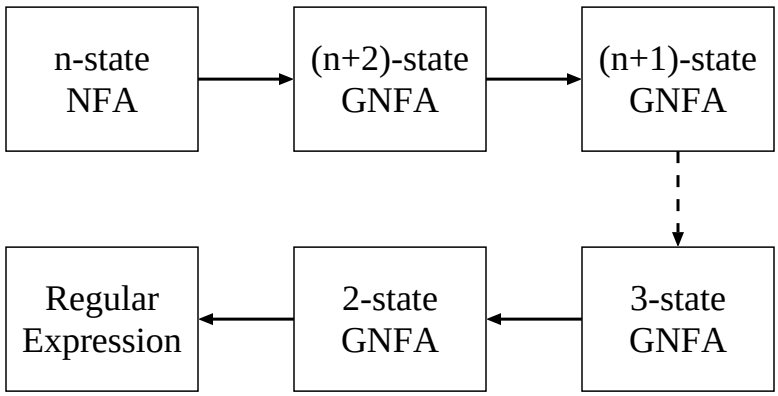

Figure 2: Steps in the algorithm to convert an NFA M to an equivalent regular expression R.

routes, but since we only want to remove state s, those do not matter. When removing state s from the machine, the route from q_i to q_j via s ceases to exist, so words accepted along that route will no longer be accepted. To fix this, we can add the regular expression describing all words accepted along that route to the direct transition from q_i to q_j: $\delta'(q_i, q_j) := R_4 \cup (R_1 R_2{}^* R_3)$, see Figure 3. If we perform this fixing step for all possible pairs of states $(q_i, q_j) \in \big(Q \setminus \{s, q_{\text{final}}\}\big) \times \big(Q \setminus \{s, q_{\text{start}}\}\big)$, including those where $q_i = q_j$, we will have included all possible ways the computation could have proceeded through the now missing state s into some other transitions. This way, the reduced machine will accept the same language as the old one.

Here, again, is the algorithm to reduce the number of states of machine $M = (Q, \Sigma, \delta, q_{\text{start}}, q_{\text{final}})$ by one:

1. Pick a state $s \in Q \setminus \{q_{\text{start}}, q_{\text{final}}\}$.
2. For every pair of states $(q_i, q_j) \in \big(Q \setminus \{s, q_{\text{final}}\}\big) \times \big(Q \setminus \{s, q_{\text{start}}\}\big)$, define
$$\delta'(q_i, q_j) := \delta(q_i, q_j) \cup \big(\delta(q_i, s)\delta(s, s)^*\delta(s, q_j)\big)$$
3. Then define the new machine $M' := \big(Q \setminus \{s\}, \Sigma, \delta', q_{\text{start}}, q_{\text{final}}\big)$

3.2 Conversion Example

As an example of a conversion from an NFA to a regular expression, we consider the automaton M shown in Figure 4, top row, defined by the shorthand

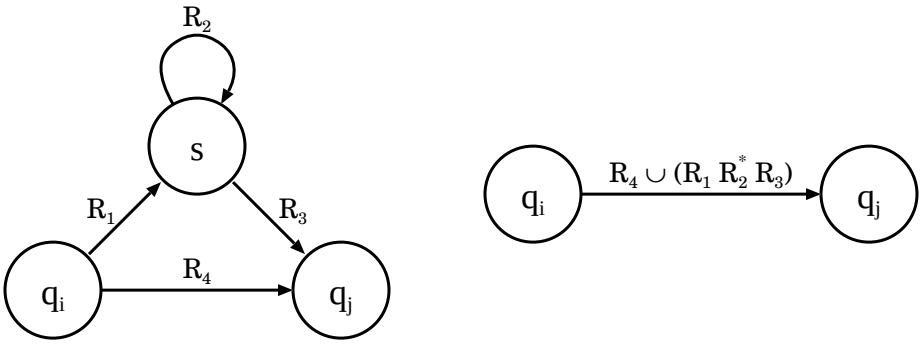

Figure 3: Fixing the transition function upon removing state s from the machine. Left: A fragment of the transition diagram consisting of two states q_i and q_j and the to be removed state s. Right: The new transition diagram after removal of s.

formal notation

$$M := \begin{array}{r|ccc} \delta & \mathtt{a} & \mathtt{b} & \epsilon \\ \hline \longrightarrow q_0 & \{q_0\} & \{q_1\} & \emptyset \\ * \, q_1 & \{q_0\} & \{q_1\} & \emptyset \end{array}$$

To convert this NFA (a DFA, in fact), we first have to add another start state and another final state. The resulting GNFA M_4, shown in Figure 4, second row, is described by $M_4 = \big(\{s, q_0, q_1, f\}, \Sigma, \delta_4, s, f\big)$, where δ_4 is given by the table

δ_4	q_0	q_1	f
s	ϵ	$\emptyset$	$\emptyset$
q_0	$\mathtt{a}$	$\mathtt{b}$	$\emptyset$
q_1	$\mathtt{a}$	$\mathtt{b}$	ϵ

Next we eliminate state q_0, yielding the GNFA M_3 shown in Figure 4, third row, defined by $M_3 = \big(\{s, q_1, f\}, \Sigma, \delta_3, s, f\big)$, where δ_3 is given by the table

δ_3	q_1	f
s	$\mathtt{a}^*\mathtt{b}$	$\emptyset$
q_1	$\mathtt{b} \cup \mathtt{aa}^*\mathtt{b}$	ϵ

Finally, we remove state q_1, yielding the 2-state GNFA M_2 shown in Figure 4, bottom row, defined by $M_2 = \big(\{s, f\}, \Sigma, \delta_2, s, f\big)$, where δ_2 is given by the table

δ_2	f
s	$\mathtt{a}^*\mathtt{b}(\mathtt{b} \cup \mathtt{aa}^*\mathtt{b})^*$

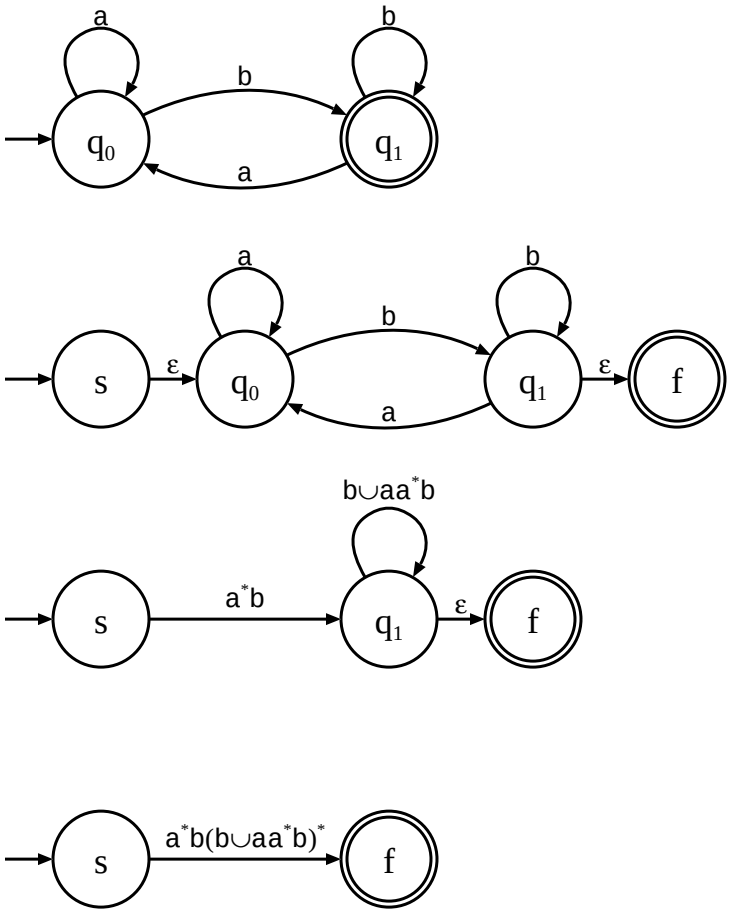

Figure 4: The steps in converting an NFA to an equivalent regular expression. Top row: the original NFA; second row: The equivalent GNFA; third row: the equivalent GNFA after removing state q_0; bottom row: the equivalent 2-state GNFA after removing state q_1. The transition label on the only transition from s to f in the 2-state GNFA is the equivalent regular expression.

The only transition in M_2 is labeled with the regular expression $R = \mathtt{a}^*\mathtt{b}(\mathtt{b} \cup \mathtt{aa}^*\mathtt{b})^*$, which is by our construction equivalent to the original NFA M.