CS 361 Meeting 7 — 2/24/20

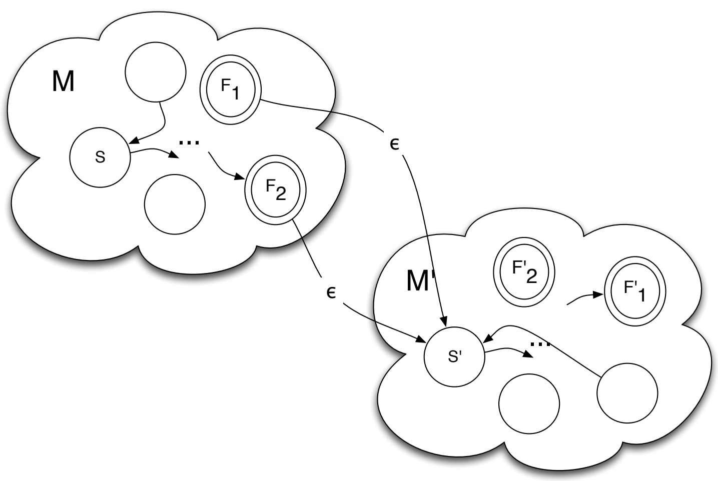
Announcements

1. Homework 3 is now online.

More Fun with NFAs

1. We can use the notion of an NFA to give some simple proofs of a few important closure properties.

   - In an earlier class, I sketched how NFAs could be used to show that regular languages are closed under the product operation.
     - Then, I suggested somehow merging the final states of a machine that recognzied one language with the start state of a second machine.
     - Now, we can use the magic of ϵ-transitions to connect the appropriate states.



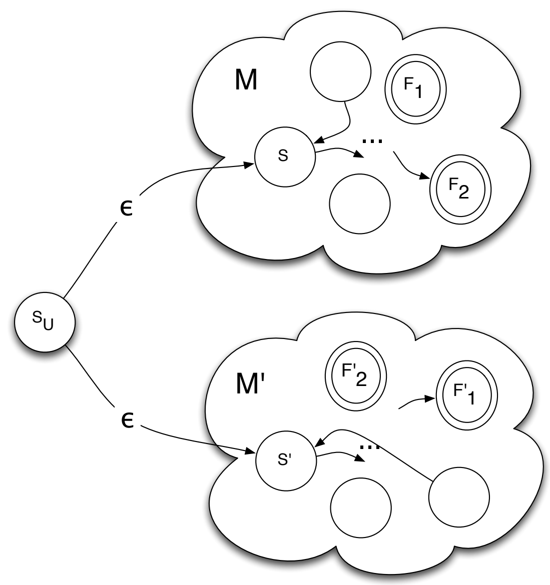
     - It is worth noting that this can be formalized by saying that if $M = (Q, \Sigma, \delta, s, F)$ and $M' = (Q', \Sigma, \delta', s', F')$ are DFAs that accept L and L' then $LL' = L(M_P)$ where the NFA M_P is defined as:

$$M_P = (Q \cup Q', \Sigma, \delta_P, s, F')$$

with

* $\delta_P(q, \epsilon) = \{s'\}$ if $q \in F$
* $\delta_P(q, x) = \{\delta(q, x)\}$ if $q \in Q$
* $\delta_P(q, x) = \{\delta'(q, x)\}$ if $q \in Q'$

   - We can also use NFAs to give a simpler proof that regular languages are closed under union.
     - We can build a new machine by taking all of the states and transitions of two machines that recognize the languages we are intersecting and adding a new state that will be our start state and have epsilon-transitions to the start states of the sub-machines so that our machine can guess which of the two languages its input belongs to and take the appropriate epsilon-transition into that machine's start state.



Click here to view the slides for this class

- Technically, we would argue that if $M = (Q, \Sigma, \delta, s, F)$ and $M' = (Q', \Sigma, \delta', s', F')$ are DFAs that accept L and L' then $L \cup L' = L(M)$ where the NFA M is defined by:

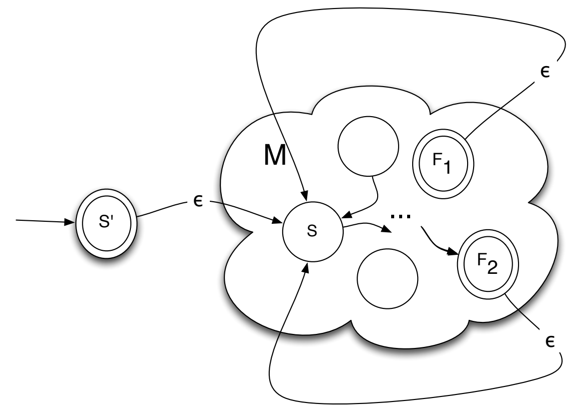
$$M_U = (Q \cup Q' \cup s_U, \Sigma, \delta_U, s_U, F \cup F')$$

  with
    - $\delta_U(s_U, \epsilon) = \{s, s'\}$
    - $\delta_U(q, x) = \{\delta(q, x)\}$ if $q \in Q$
    - $\delta_U(q, x) = \{\delta'(q, x)\}$ if $q \in Q'$

- Regular languages are closed under the closure (star) operation.
  - Recall that when we write L^n we mean $\{\epsilon\}$ if $n = 0$ and $L^n = LL^{n-1}$ otherwise and that we define

$$L^* = \bigcup_{n=0}^{\infty} L^n$$

  - At first, it might seem obvious that the regular languages are closed under the closure operation (both because of its name and) because the closure is a union of products and we have already shown that regular languages are closed under union and products, but...
  - We have shown that regular languages are closed under *finite* unions. This does not imply they are closed under infinite unions.
    - Every language containing just a single word is regular. Therefore, if the regular languages were closed under infinite unions, we could show that any language was regular by just union-ing together all of the languages consisting of just one word from the possibly non-regular language.
    - It is also worth noting that if we think about the way we proved unions of regular languages were regular, applying the construction to an infinite set of finite automata would lead to an infinite automaton.
  - We can, however, show that the closure of a regular language is regular using a construction involving NFAs with epsilon transitions. Given a machine M that recognized the language we want the closure of, the idea is to add a new, final start state (to handle the fact that ϵ must be in the closure), allow an epsilon-transition from this state to the original start state, and also add epsilon-transitions from all of the final states back to the original start state.



  - That is, iv $M = (Q, \Sigma, \delta, s, F)$ is a DFA that accept L then we claim that $L^* = L(M_C)$ where the NFA M_C is defined as:

$$M_C = (Q \cup \{s_C\}, \Sigma, \delta_C, s_C, F \cup s_C)$$

  with
    - $\delta_C(q, \epsilon) = \{s\}$ if $q \in F$
    - $\delta_C(s_C, \epsilon) = \{s\}$
    - $\delta_C(q, x) = \{\delta(q, x)\}$ if $q \in Q$

  accepts L^*

String Ninjas

1. Let's look at an example that takes full advantage of the power of non-determinism.

2. Suppose that we define

$$L_{\frac{1}{2}} = \{x \,|\, \text{there exists } y \text{ such that } |x| = |y| \text{ and } xy \in L\}$$

   Consider how we can prove that $L_{\frac{1}{2}}$ is regular if L is regular.

3. I want to explore two distinct approaches to building NFAs that show that regular languages are closed under this operation.

    - One approach is to build an NFA that simultaneously simulates a DFA examining its input in two directions at the same time. One copy of the simulated DFA starts at the beginning of the actual input. The other simulated copy works its way backward from the end of an imagined second-half of the input. The goal of the backward simulation is to guess a string that could serve as the y in the definition of $L_{\frac{1}{2}}$
    - The other approach still manages two simultaneous simulations, but both run forward. One scans the actual input. The other guesses and scan a string that might form y from a state that the machine also guesses will correspond to $\hat{\delta}(s, x)$.
    - In both of these simulations, we will use non-determinism to guess things (all the letters of y and either the machine state that comes between x and y or the final state after y). We will, however, carefully define our final states so that they will only be reachable if the machine makes the right guesses.

    This illustrates why the power to guess is actually a reasonable feature to include in the NFA model. NFAs can guess, but they also have to check that there guesses were right!

4. Let's consider the parallel forward scan version:

    - Suppose that D is a DFA that accepts L.
    - The idea is that given input $x = x_1x_2 \ldots x_n$, we want so simulate one version of D scanning x at the same time we simulate another version of D scanning some string $y = y_1y_2 \ldots y_n$ in the hope of verifying that $xy \in L(D)$.
    - This involves a lot of guessing. Most obviously, the second version of D we simulate has to guess what all the y_is are.
    - In addition, while we want the simulation of the scan of x to start in the start state, s of D, that is not where the scan of y should start. Instead, we should start the scan of y in the state $m = \hat{\delta}(s, x)$. Our first scan will figure out what m is. Unfortunately, it will not have figured this out yet when we need to start the simulated scan of y. So, we also need to guess m!
    - The trick that makes all this guessing work is that we will design N to verify that all our guesses were correct. In this case, there are two things we must verify when both scans are finished:
        - We have to verify that the state we reach at the end of scanning x is equal to the m we guessed as we started scanning y. This verifies our machine's guess of m.
        - We need to verify that the scan of y from state m ends in a final state. This verifies that the machine guessed the letters of y correctly.
    - To make all this formal, we have to specify the 5 components of an NFA.
        - The new machine will use the same alphabet, Σ, as the machine it will simulate.
        - We need to be able to remember which state each of our scans is in while also remembering the state m we guessed at the beginning so that we can verify it at the end. Therefore, our states must hold three states of the simulated machine so $Q \times Q \times Q \subset Q_N$. The first state in each triple will hold the state of the scan of x, the second state will be the value of m, and the third state will be the current state of the scan of y.
        - As our start state we will introduce a new state s_N from which ϵ-transitions can be used to make our guess of m. This is the only extra state we need so the complete set of states will be $\{s_N\} \cup Q \times Q \times Q$.
        - To be successful, our simulation must reach a state where the scan of x ends in m and the scan of y ends in a final state of D. So $F_N = \{(m, m, f) \mid m \in Q, f \in F\}$.
        - The initial guess of m is accomplished by including the following ϵ-transitions in the definition of δ:

        $$\delta(s_n, \epsilon) = \{(s, m, m) \mid m \in Q\}$$

        - For all other states, $\delta(s_n, \epsilon) = \emptyset$.

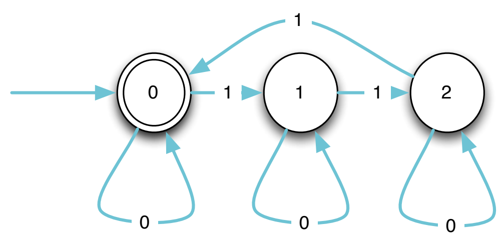
– Finally, the simulated scans proceed using the transitions

$$\delta((s_x, m, s_y), x_i) = \{(\delta(s_x, x_i), m, \delta(s_y, y_i)) \mid y_i \in \Sigma\}$$

Here, s_x refers to the state of the simulated version of the machine scanning the actual input that corresponds to "x" in the definition of $L_{\frac{1}{2}}$, s_y is the state of the simulated scan of the guessed string y. x_i is the next input symbol. y_i is the next symbol guessed to be in the string y.

5. Before leaving this fun example, it is worth discussing one concrete example of the application of the abstract transformation from a particular D to a particular N the transformation just described would produce.

   - Consider the machine D shown below which recognizes the language of sequences of 0s and 1s containing a number of 1s that is a multiple of 3.

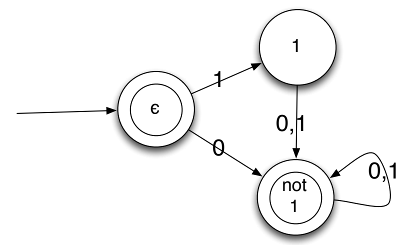


   - First, it is worth asking what language we get when we apply the $L_{\frac{1}{2}}$ transformation to the language of this machine without thinking about how a machine to accept it might be formed.

     – Given any string of length greater than 1, we can calculate the number of 1s in the string mod 3. If the answer is 2 we add a single 1, if the answer is 1, we add two 1s, and if the answer is 0, we leave it alone for a moment. Then, regardless of how many 1s we just added, we add enough 0s to double the length of the original string. The result belongs to $L(D)$! So, any string longer than 1 belongs to $L(D)_{\frac{1}{2}}$.

     – In fact, the only string that is not in $L(D)_{\frac{1}{2}}$ is "11".

   - It is pretty clear that it would be easy to build a machine to accept this language. One possibility is shown below.



   - Interesting, however, the machine produce by our transformation will have describing $N_{\frac{1}{2}}$ will have $3^3 + 1 = 28$ states and a very complicated set of transitions connecting these states.

Regular Expressions

1. The closure properties of regular languages provide a way to describe regular languages by building them out of simpler regular languages using the operations union, product and closure.

2. The notation called *regular expressions* is based on this fact.

   Definition: Given some finite alphabet Σ, we define e to be a regular expression if e is

   - a for some $a \in \Sigma$
   - $\emptyset$
   - ε
   - $e_0 \cup e_1$, where e_0 and e_1 are regular expressions
   - $e_0 \circ e_1 = e_0 e_1$ where e_0 and e_1 are regular expressions
   - e_0^* where e_0 is a regular expression.
   - (e_0) where e_0 is a regular expression.

3. We view regular expressions as another formalism for describing languages. If e is a regular expression, the language defined by e is denoted by $L(e)$ and defined recursively/inductively as follows:

   Base clauses

- $L(x)$ for some $a \in \Sigma$ is just $\{a\}$
- $L(\emptyset)$ is $\emptyset$
- $L(\varepsilon)$ is $\{\epsilon\}$

Recursive clauses

- $L(e_0 \cup e_1)$ is $L(e_0) \cup L(e_1)$
- $L(e_0 \circ e_1)$ is $L(e_0)L(e_1)$
- $L(e_0^*)$ is $L(e_0)^*$
- $L((e_0))$ is $L(e_0)$

4. Given the closure properties we have just shown, it is clear that all regular expressions describe regular languages.