Finite Automata and Formal Languages

TMV026/DIT321– LP4 2012

Lecture 9

Ana Bove

April 17th 2012

Overview of today's lecture:

- Closure Properties for Regular Languages
- Decision Properties for Regular Languages

More Closure Properties for Regular Languages

We shall now see that RL are also closed under the following operations:

- Reversal
  Recall that intuitively, $\text{rev}(a_1 \ldots a_n) = a_n \ldots a_1$ (slide 13, lecture 3) and that $\forall x, \text{rev}(\text{rev}(x)) = x$ (slide 14, lecture 3)

  Given $\mathcal{L}$, let $\mathcal{L}^{\text{r}} = \{\text{rev}(x) \mid x \in \mathcal{L}\}$;

- Homomorphism (substitution of string by symbols);

- Inverse homomorphism.

Closure under Reversal

We define the following function over RE:

$$\emptyset^{\mathsf{r}} = \emptyset \qquad \epsilon^{\mathsf{r}} = \epsilon \qquad a^{\mathsf{r}} = a$$
$$(R_1 + R_2)^{\mathsf{r}} = R_1^{\mathsf{r}} + R_2^{\mathsf{r}}$$
$$(R_1 R_2)^{\mathsf{r}} = R_2^{\mathsf{r}} R_1^{\mathsf{r}}$$
$$(R^*)^{\mathsf{r}} = (R^{\mathsf{r}})^*$$

Theorem: *If $\mathcal{L}$ is regular so is $\mathcal{L}^{\mathsf{r}}$.*

Proof: (See theo. 4.11, pages 139–140). Let R be a RE such that $\mathcal{L} = \mathcal{L}(R)$.
We need to prove by structural induction on R that $\mathcal{L}(R^{\mathsf{r}}) = (\mathcal{L}(R))^{\mathsf{r}}$.
Hence $\mathcal{L}^{\mathsf{r}} = (\mathcal{L}(R))^{\mathsf{r}} = \mathcal{L}(R^{\mathsf{r}})$ and $\mathcal{L}^{\mathsf{r}}$ is regular.

Example: The reverse of the language defined by $(0+1)^*0$ can be defined by $0(0+1)^*$.

Closure under Reversal

Another way to prove this result is by constructing a ϵ-NFA for $\mathcal{L}^{\mathsf{r}}$.

Proof: Let $N = (Q, \Sigma, \delta_N, q_0, F)$ be a NFA such that $\mathcal{L} = \mathcal{L}(N)$.
Define $E = (Q \cup \{q\}, \Sigma, \delta_E, q, \{q_0\})$ with $q \notin Q$ and δ_E such that

$$r \in \delta_E(s, a) \text{ iff } s \in \delta_N(r, a) \text{ for } r, s \in Q$$
$$r \in \delta_E(q, \epsilon) \text{ iff } r \in F$$

Recall: Functions between Languages

(from slide 21, lecture 3)

Definition: A function $f : \Sigma^* \to \Delta^*$ between 2 languages should be such that it satisfies

$$f(\epsilon) = \epsilon$$
$$f(xy) = f(x)f(y)$$

Intuitively, $f(a_1 \ldots a_n) = f(a_1) \ldots f(a_n)$.
Notice that $f(a) \in \Delta^*$ if $a \in \Sigma$.

Definition: f is called *coding* iff f is *injective*.

Definition: $f(\mathcal{L}) = \{f(x) \mid x \in \mathcal{L}\}$.

Languages are Monoids

Definition: A *monoid* is an algebraic structure with an associative binary operation and an identity element.

Let Σ be an alphabet.
Then Σ^* is a monoid if we consider the concatenation as binary operation and ϵ as the identity element with respect to the binary operation.

Recall:

- Concatenation is associative: $(xy)z = x(yz)$
- $x\epsilon = \epsilon x = \epsilon$
- Concatenation is in general not commutative (but this is not required in the definition of a monoid)

Homomorphisms

Definition: A *homomorphism* is a structure-preserving map between 2 algebraic structures.

Note: A function $h : \Sigma^* \to \Delta^*$ satisfying

$$h(\epsilon) = \epsilon$$
$$h(xy) = h(x)h(y)$$

can be seen as a homomorphism between the monoids (languages) Σ^* and Δ^*.

Recall we have then that $h(a_1 \ldots a_n) = h(a_1) \ldots h(a_n)$.

Closure under Homomorphisms

Theorem: *If $\mathcal{L}$ is a RL over Σ and $h : \Sigma^* \to \Delta^*$ is an homomorphism on Σ then $h(\mathcal{L})$ is also regular.*

Proof: We define the following function over RE:

$$f_h(\emptyset) = \emptyset \qquad f_h(\epsilon) = \epsilon \qquad f_h(a) = h(a)$$
$$f_h(R_1 + R_2) = f_h(R_1) + f_h(R_2)$$
$$f_h(R_1 R_2) = f_h(R_1) f_h(R_2)$$
$$f_h(R^*) = (f_h(R))^*$$

We need to prove by structural induction on R that $\mathcal{L}(f_h(R)) = h(\mathcal{L}(R))$.
Now, if $\mathcal{L} = \mathcal{L}(R)$ then we have that $h(\mathcal{L})$ is regular since
$h(\mathcal{L}) = h(\mathcal{L}(R)) = \mathcal{L}(f_h(R))$.
(See Theorem 4.14, pages 141–142.)

Closure under Homomorphisms

Let $h : \Sigma^* \rightarrow \Delta^*$ be a homomorphism and $\mathcal{L}$ a RL over Σ.

By the previous theorem and the definition of RL, we know that there exists a DFA D over Σ and a DFA F over Δ such that

$$\mathcal{L} = \mathcal{L}(D) \quad \text{and} \quad h(\mathcal{L}) = \mathcal{L}(F)$$

F can be constructed from the RE for $\mathcal{L}$ (via an ϵ-NFA).

Often not obvious how to construct the DFA directly.

Inverse Homomorphisms

Definition: If $h : \Sigma^* \rightarrow \Delta^*$ is a homomorphism and $\mathcal{L}$ is a language over Δ, $h^{-1}(\mathcal{L})$ (read *h inverse of $\mathcal{L}$*) is the set of strings w such that $h(w) \in \mathcal{L}$.
In other words, $h^{-1}(\mathcal{L}) = \{w \in \Sigma^* \mid h(w) \in \mathcal{L}\}$.

Note: h^{-1} does not necessarily correspond to a function!

Example: Imagine we have that $h(a) = c$, $h(b) = c$ and $\mathcal{L} = \{c\}$.
Then $h^{-1}(\mathcal{L}) = \{a, b\}$ but h^{-1} itself is not a function.

Closure under Inverse Homomorphisms

Theorem: *Let $h : \Sigma^* \to \Delta^*$ be a homomorphism. If $\mathcal{L}$ is a RL over Δ then $h^{-1}(\mathcal{L})$ is a RL over Σ.*

Proof: Let $D = (Q, \Delta, \delta, q_0, F)$ be a DFA such that $\mathcal{L} = \mathcal{L}(D)$.
We define the DFA $D' = (Q, \Sigma, \delta', q_0, F)$ over Σ such that

$$\delta'(q, a) = \hat{\delta}(q, h(a))$$

By induction on $|w|$ we prove that $\hat{\delta}'(q, w) = \hat{\delta}(q, h(w))$
(Recall that $\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y)$.)
Then D' accepts w iff D accepts $h(w)$ (since the set of accepting states is the same in both DFA).

Note: Since h^{-1} might not be a function it seems difficult to directly define the RE that corresponds to the h inverse of $\mathcal{L}$.

Example: $\mathcal{L}'$ from Slide 14 Lecture 8

Example: We know $\mathcal{L} = \{b^m c^m \mid m \geqslant 0\}$ is not regular.
Let us consider $\mathcal{L}' = a^+\mathcal{L} \cup (b + c)^*$.

We will prove that $\mathcal{L}'$ is not regular. Let us assume it is.

Then $a^+\mathcal{L} = \mathcal{L}' \cap \overline{(b + c)^*}$ must be regular.

Then, $\mathcal{L} = h(a^+\mathcal{L})$ must also be regular, where h is the following homomorphism: $h(a) = \epsilon, h(b) = b, h(c) = c$.

We arrive at a contradiction, hence $\mathcal{L}'$ cannot be regular.

Decision Properties of Regular Languages

We want to be able to answer YES/NO to questions such as

- Is this language empty?
- Is string w in the language $\mathcal{L}$?
- Are these 2 languages equivalent?

In general languages are infinite so we cannot do a "manual" checking.

Instead we should work with the finite description of the languages (DFA, NFA. ϵ-NFA, RE).
Which description is the most convenient depends on the property and on the language.

Testing Emptiness of Regular Languages

Given a FA for a language, testing whether the language is empty or not amounts to checking if there is a path from the start state to a final state.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA.

Recall the notion of accessible states from slide 22 in lecture 4:

Definition: The set $\text{Acc} = \{\hat{\delta}(q_0, x) \mid x \in \Sigma^*\}$ is the set of *accessible* states (from the state q_0).

Proposition: *Given D as above, then $D' = (Q \cap \text{Acc}, \Sigma, \delta', q_0, F \cap \text{Acc})$, where δ' is the function δ restricted to the states in $Q \cap \text{Acc}$, is a DFA such that $\mathcal{L}(D) = \mathcal{L}(D')$.*

In particular, $\mathcal{L}(D) = \emptyset$ if $F \cap \text{Acc} = \emptyset$.
(Actually, $\mathcal{L}(D) = \emptyset$ iff $F \cap \text{Acc} = \emptyset$ since if $\hat{\delta}(q_0, x) \in F$ then $\hat{\delta}(q_0, x) \in F \cap \text{Acc}$.)

Testing Emptiness of Regular Languages

A recursive algorithm to test whether a state is accessible/reachable is as follows:

Base case: The start state q_0 is reachable from q_0.

Recursive step: If q is reachable from q_0 and there is an arc from q to p (with any label, including ϵ) then p is also reachable from q_0.

(This algorithm is an instance of *graph-reachability*.)

If the set of reachable states contains at least one final state then the RL is NOT empty.

Functional Representation of Testing Emptiness for FA

```
import List(union)

data Q = ...   deriving Eq

data S = ...

final :: Q -> Bool

delta :: Q -> S -> Q

isIn :: [Q] -> Q -> Bool
isIn = flip elem

isSuperSet :: [Q] -> [Q] -> Bool
isSuperSet as bs = and (map (isIn as) bs)
```

Functional Representation of Testing Emptiness for FA

The first argument in the functions below is a list with *all* symbols in the `S`.

```
closure :: [S] -> (Q -> S -> Q) -> [Q] -> [Q]
closure cs delta qs =
       let qs' = qs >>= (\q -> map (delta q) cs)
       in if isSuperSet qs qs' then qs
          else closure cs delta (union qs qs')
```

```
accessible :: [S] -> (Q -> S -> Q) -> Q -> [Q]
accessible cs delta q = closure cs delta [q]
```

```
notEmpty :: [S] -> (Q -> S -> Q) -> Q -> Bool
notEmpty cs delta q0 =
                  or (map final (accessible cs delta q0))
```


Functional Representation of Testing Emptiness for FA

The closure function can be optimised by not computing the closure of the same state twice.

```
closure :: [S] -> (Q -> S -> Q) -> [Q] -> [Q]
closure cs delta qs = clos [] qs
  where
    clos :: [Q] -> [Q] -> [Q]
    clos qs1 qs2 =
         if qs2 == [] then qs1
         else let qs = union qs1 qs2
                  qs' = qs2 >>= (\q -> map (delta q) cs)
                  qs'' = filter (\q -> not (isIn qs q)) qs'
              in clos qs qs''
```

Testing Emptiness of Regular Languages (Again)

Given a RE for the language we can instead perform the following test:

Base cases: $\emptyset$ denotes the empty language while ϵ and a (any symbol from the alphabet) do not.

Inductive step: Let R be our RE.

- If $R = R_1 + R_2$ then $\mathcal{L}(R)$ is empty iff both $\mathcal{L}(R_1)$ and $\mathcal{L}(R_2)$ are empty;
- If $R = R_1 R_2$ then $\mathcal{L}(R)$ is empty iff either $\mathcal{L}(R_1)$ or $\mathcal{L}(R_2)$ is empty;
- If $R = R_1^*$ is never empty since it always contains the word ϵ.

Functional Representation of Testing Emptiness for RE

```
data RExp a = Empty | Epsilon | Atom a |
              Plus (RExp a) (RExp a) |
              Concat (RExp a) (RExp a) |
              Star (RExp a)
```

```
isEmpty :: RExp a -> Bool
isEmpty Empty = True
isEmpty (Plus e1 e2) = isEmpty e1 && isEmpty e2
isEmpty (Concat e1 e2) = isEmpty e1 || isEmpty e2
isEmpty _ = False
```

Testing Membership in Regular Languages

Given a RL $\mathcal{L}$ and a word w over the alphabet of $\mathcal{L}$, is $w \in \mathcal{L}$?

When $\mathcal{L}$ is given by a FA we can simply run the FA with the input w and see if the word is accepted by the FA.
We have seen algorithms that simulate the running of a FA (see slides 10–11 in lecture 4 for DFA, slides 10–12 in lecture 5 for NFA, and slides 15, 18–19 in lecture 6 for ϵ-NFA).

Using *derivatives* (see exercises 4.2.3 and 4.2.5) there is a nice algorithm checking membership on RE.
Let $\mathcal{L} = \mathcal{L}(R)$ and $w = a_1 \ldots a_n$.

Let $a \backslash R = D_a R = \{x \mid ax \in \mathcal{L}\}$ (in the book $\frac{d\mathcal{L}}{da}$).

$D_w R = D_{a_n}(\ldots(D_{a_1} R)\ldots)$.
It can then be shown that $w \in \mathcal{L}$ iff $\epsilon \in D_w R$.

Other Testing Algorithms on Regular Expressions

Tests if a RE contains ϵ.

```
hasEpsilon :: RExp a -> Bool
hasEpsilon Epsilon = True
hasEpsilon (Star _) = True
hasEpsilon (Plus e1 e2) = hasEpsilon e1 || hasEpsilon e2
hasEpsilon (Concat e1 e2) = hasEpsilon e1 && hasEpsilon e2
hasEpsilon _ = False
```

Other Testing Algorithms on Regular Expressions

Tests if $\mathcal{L}(R) \subseteq \{\epsilon\}$.

```
atMostEps :: RExp a -> Bool
atMostEps Empty = True
atMostEps Epsilon = True
atMostEps (Atom _) = False
atMostEps (Plus e1 e2) = atMostEps e1 && atMostEps e2
atMostEps (Concat e1 e2) = isEmpty e1 || isEmpty e2 ||
                           (atMostEps e1 && atMostEps e2)
atMostEps (Star e) = atMostEps e
```


Other Testing Algorithms on Regular Expressions

Tests if a regular expression denotes an infinite language.

```
infinite :: RExp a -> Bool
infinite (Star e) = not (atMostEps e)
infinite (Plus e1 e2) = infinite e1 || infinite e2
infinite (Concat e1 e2) = (infinite e1 && notIsEmpty e2) ||
                          (notIsEmpty e1 && infinite e2)
  where notIsEmpty e = not (isEmpty e)
infinite _ = False
```