Finite Automata and Formal Languages

TMV026/DIT321 – LP4 2010

Lecture 8

April 22th 2010

Overview of today's lecture:

- **Decision Properties of Regular Languages**

Decision Properties of Regular Languages

We want to be able to answer 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 14 in lecture 3:

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

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

In particular, $\mathcal{L}(D) = \emptyset$ if $F \cap \mathsf{Acc} = \emptyset$.

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

Testing Emptiness of Regular Languages

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

Basis: The start state q_0 is reachable from the q_0.

Inductive 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

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

Basis: $\emptyset$ 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 slide 11 in lecture 3 for DFA, slides 10–13 in lecture 4 for NFA, and slides 16 and 19–20 in lecture 5 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 $w = a_1 \dots a_n$.

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

$D_w R = D_{a_n}(\dots(D_{a_1} R)\dots)$.

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

Test 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
```