COMPSCI 250: Introduction to Computation

Lecture #30: Properties of the Regular Languages

David Mix Barrington and Ghazaleh Parvini

13 November 2023

Properties of Regular Languages

- Induction on Regular Expressions
- The One’s Complement Operation
- Proving Our Function Correct
- The Pseudo-Java RegExp Class
- The One’s Complement Method
- Reversal of Languages
- Testing for the Empty Language

Induction on Regular Expressions

- Because the regular languages have an inductive definition, we can prove propositions for all of them by induction.
- Let P(R) be a predicate with one free variable of type "regular expression". We can prove that P(R) holds for any regular expression R by proving *two* base cases and *three* inductive cases.

Induction on Expressions

- These five cases are:
- $P(\emptyset)$,
- $P(a)$ for all $a \in \Sigma$,
- $(P(R) \wedge P(S)) \rightarrow P(R + S)$,
- $(P(R) \wedge P(S)) \rightarrow P(RS)$, and
- $P(R) \rightarrow P(R^*)$

Induction on Expressions

- For example, we will define two operations on languages and show that the regular languages are **closed** under these operations.
- That is, if R is a regular expression, the result of applying the operation to L(R) gives us another regular language. We'll demonstrate an algorithm to compute this expression.
- We'll also show that we can test properties of R, such as whether $L(R) = \emptyset$.

One's Complement

- The **one's complement** of a binary string w, denoted oc(w), is the string of the same length obtained by replacing all 0's with 1's and all 1's with 0's. For example, oc(011001) = 100110.
- We can define oc(w) inductively, of course:
- $oc(\lambda) = \lambda$,
- $oc(w0) = oc(w)1$, and
- $oc(w1) = oc(w)0$.

One's Complement

- The one's complement of a *language* X is the language {oc(w): w $\in$ X} -- the set of strings whose one's complements are in X.
- We will prove that for any regular expression R, the language oc(L(R)) is a regular language.
- It's not hard to see how to convert R into a regular expression for oc(L(R)). We just replace 0's with 1's and 1's with 0's in R itself.

One's Complement

- Formally this is a recursive algorithm:
- $oc(\emptyset) = \emptyset$,
- $oc(0) = 1$,
- $oc(1) = 0$,
- $oc(R + S) = oc(R) + oc(S)$,
- $oc(RS) = oc(R)oc(S)$, and
- $oc(R^*) = oc(R)^*$.

Proving Our Function Correct

- We'll use induction to prove that this function f, from regular expressions to regular expressions, satisfies the property "$L(f(R)) = oc(L(R))$".
  We write this property as "$P(R)$".
- $P(\emptyset)$ says that $L(\emptyset) = oc(L(\emptyset))$, which is true because $\{oc(w): w \in \emptyset\} = \emptyset$.
- $P(0)$ says "$L(1) = oc(L(0))$" and $P(1)$ says "$L(0) = oc(L(1))$". Both are true.

Proving Our Function Correct

- Assume that $P(R)$ and $P(S)$ are true, so that $L(f(R)) = oc(L(R))$ and $L(f(S)) = oc(L(S))$.
- We must show
  that $L(f(R)) \cup L(f(S)) = oc(L(R+S))$,
  that $L(f(R))L(f(S)) = oc(L(RS))$,
  and that $L(f(R))^* = oc(L(R^*))$.
- Each of these three facts follow pretty directly from the definitions -- details are in the textbook.

Clicker Question #1

- To formally prove the statement “$oc(S^*) = oc(S)^*$”, I am trying to prove that “$oc(S^*)$ is a subset of $oc(S)^*$”. Which *would* be a good inductive step of my proof?
- (a) $((u \in oc(S)^*) \wedge (v \in oc(S)^*) \rightarrow uv \in oc(S)^*$
- (b) $((u \in S^*) \wedge (v \in S)) \rightarrow uv \in oc(S)^*$
- (c) $((u \in oc(S^*)) \wedge (v \in oc(S)) \rightarrow uv \in oc(S^*)$
- (d) $((u \in oc(S^*)) \wedge (v \in oc(S)) \rightarrow uv \in oc(S)^*$

Not the Answer

Clicker Answer #1

- To formally prove the statement "$oc(S^*) = oc(S)^*$", I am trying to prove that "$oc(S^*)$ is a subset of $oc(S)^*$". Which *would* be a good inductive step of my proof?
- (a) $((u \in oc(S)^*) \wedge (v \in oc(S)^*) \to uv \in oc(S)^*$
- (b) $((u \in S^*) \wedge (v \in S)) \to uv \in oc(S)^*$
- (c) $((u \in oc(S^*)) \wedge (v \in oc(S)) \to uv \in oc(S^*)$
- *(d) $((u \in oc(S^*)) \wedge (v \in oc(S)) \to uv \in oc(S)^*$*

More Answer #1

- Goal is to prove "$oc(S^*) \subseteq oc(S)^*$".
- (a) $((u \in oc(S)^*) \wedge (v \in oc(S)^*) \rightarrow uv \in oc(S)^*$
  This is true, but says nothing about $oc(S^*)$
- (b) $((u \in S^*) \wedge (v \in S)) \rightarrow uv \in oc(S)^*$ no reason to think that this is true
- (c) $((u \in oc(S^*)) \wedge (v \in oc(S)) \rightarrow uv \in oc(S^*)$
  this gets the wrong conclusion for our goal
- *(d)* $((u \in oc(S^*)) \wedge (v \in oc(S)) \rightarrow uv \in oc(S)^*$

A Java `RegExp` Class

- Just as boolean or arithmetic expressions can be implemented by tree structures, we can define a real Java class `RegExp` whose objects are regular expressions.
- We will need methods to **parse** these objects, which means that they must determine their structure and component parts.

A Java `RegExp` Class

- `public class RegExp {`

```
  public RegExp( );
    // returns RegExp equal to emptyset
  public RegExp(String w);
    // returns RegExp denoted by w
  public boolean isEmptySet( );
    // is it the empty set?
  public boolean isZero( );
    // is it “0”?
  public boolean isOne( );
    // is it “1”?
  public boolean isUnion( );
    // is it “S + T”?
```

A Java RegExp Class

```
public boolean isCat( );
  // is it “ST”?
public boolean isStar( );
  // is it “S*”?
public RegExp firstArg( );
public RegExp secondArg( );
public static RegExp
  plus (RegExp r, RegExp s);
public static RegExp
  cat (RegExp r, RegExp s);
public static RegExp
  star (RegExp r);
```

Computing One’s Complement

- This definition lets us write code for the one’s complement algorithm. The next slide has a recursive method that creates a `RegExp` object with the same structure as the method’s argument, but with 0’s and 1’s switched.
- We’ve essentially proved this method correct by our usual method for recursive code -- we prove the base cases correct and then prove the rest correct assuming that the recursive calls are correct.

Computing One’s Complement

```
public static RegExp f (RegExp s) {
   if (s.isEmptySet( ))
      return new RegExp( );
   if (s.isZero( ))
      return new RegExp(“1”);
   if (s.isOne( ))
      return new RegExp(“0”);
   RegExp oc1 = f (s.firstArg( ));
   if (s.isStar( )) return star(oc1);
   RegExp oc2 = f (s.secondArg( ));
   is (s.isPlus( ))
      return plus (oc1, oc2);
   else return cat (oc1, oc2);}
      // s.isCat( ) must be true here
```

Reversal of Languages

- A similar function from languages to languages is reversal, based on the familiar reversal operation on strings:
for any language X, $X^R = \{w^R: w \in X\}$.

- The regular languages are closed under reversal -- we can easily see that $\varnothing^R = \varnothing$ and that $a^R = a$ for any letter a. The string rule $(xy)^R = y^R x^R$ yields a language rule
$(TU)^R = U^R T^R$, and we have $(T+U)^R = T^R + U^R$ and $(T^*)^R = (T^R)^*$.

Computing Reversal

```
public static RegExp rev (RegExp s) {
    if (s.isEmptySet( ))
        return new RegExp( );
    if (s.isZero( ))
        return new RegExp("0");
    if (s.isOne( ))
        return new RegExp("1");
    RegExp rev1 = rev (s.firstArg( ));
    if (s.isStar( )) return star (rev1);
    RegExp rev2 = rev (s.secondArg( ));
    if (s.isPlus( ))
        return plus (rev1, rev2);
    else return cat (rev2, rev1);}
        // s.isCat( ) is true in this case
```

Clicker Question #2

- For the case where s is a union, rev contains the line return `plus(rev1, rev2)`;
  What would happen if we changed this line to return `plus(rev2, rev1)`; ?
- (a) `rev` would return a different expression, but equivalent to the one it returned before
- (b) `rev` would become the identity function
- (c) `rev` would return exactly the same expression
- (d) rev would return something not equivalent to the correct reversal, and also not the identity

Not the Answer

Clicker Question #2

- For the case where s is a union, rev contains the line return `plus(rev1, rev2);`
  What would happen if we changed this line to return `plus(rev2, rev1);` ?
- *(a) `rev` would return a different expression, but equivalent to the one it returned before* for example we'd return "S+R" instead of "R+S"
- (b) `rev` would become the identity function
- (c) `rev` would return exactly the same expression
- (d) rev would return something not equivalent to the correct reversal, and also not the identity

Testing for the Empty Language

- The regular expression "$\emptyset$" denotes the empty language, but so do other regular expressions like $a(b+a)^*(\emptyset + a^*\emptyset)(bb)^*$.
- Exercise 5.5.4 asks you to write a method that takes a `RegExp` object R and returns a boolean that is true if and only if $L(R) = \emptyset$.

Testing for the Empty Language

- We solve the problem recursively.
- For the base cases, we should return `true` on $\emptyset$ and return `false` on any letter a.
- If R and S are two regular expressions, L(R + S) is empty if and only if *both* L(R) and L(S) are empty, and L(RS) is empty if and only if *either* L(R) or L(S) is empty.
- And of course $L(R^*)$ is never empty.

Testing Properties of Expressions

- A similar problem is to tell whether L(R) = {λ}, or whether $\lambda \in$ L(R). Each of these may be solved by a recursive algorithm, because we know whether the property holds in the base cases, and how it behaves with respect to the three operations.

- But telling whether L(R) = Σ^* is much harder, because L(R + S) could equal Σ^* in so many *different* ways.

Clicker Question #3

- Given a regular expression R over {a, b}, I would like to compute whether $L(R) = \{\lambda\}$. Which of these potential steps in an inductive definition of this property is *valid*?
- (a) $L(R^*) = \{\lambda\} \leftrightarrow L(R) = \emptyset$
- (b) $L(RS) = \{\lambda\} \leftrightarrow (L(R) = \{\lambda\}) \wedge (L(S) = \{\lambda\})$
- (c) $L(R+S) = \{\lambda\} \leftrightarrow (L(R) = \{\lambda\}) \wedge (L(S) = \{\lambda\})$
- (d) $L(R+S) = \{\lambda\} \leftrightarrow (L(R) = \{\lambda\}) \vee (L(S) = \{\lambda\})$

Not the Answer

Clicker Question #3

- Given a regular expression R over {a, b}, I would like to compute whether $L(R) = \{\lambda\}$. Which of these potential steps in an inductive definition of this property is *valid*?
- (a) $L(R^*) = \{\lambda\} \leftrightarrow L(R) = \varnothing$
- *(b) $L(RS) = \{\lambda\} \leftrightarrow (L(R) = \{\lambda\}) \wedge (L(S) = \{\lambda\})$*
- (c) $L(R+S) = \{\lambda\} \leftrightarrow (L(R) = \{\lambda\}) \wedge (L(S) = \{\lambda\})$
- (d) $L(R+S) = \{\lambda\} \leftrightarrow (L(R) = \{\lambda\}) \vee (L(S) = \{\lambda\})$
  for $L(R+S) = \{\lambda\}$ we need one of L(R) or L(S) to be $\{\lambda\}$, and the other to be $\{\lambda\}$ or $\varnothing$