Regular Expressions

Regular expressions can be seen as a system of notations for denoting ϵ-NFA

They form an "algebraic" representation of ϵ-NFA

"algebraic": expressions with equations such as $E_1+E_2 = E_2+E_1$ $E(E_1+E_2) = EE_1 + EE_2$

Each regular expression E represents also a language $L(E)$

Very convenient for representing pattern in documents (K. Thompson)

Regular Expressions: Abstract Syntax

Given an alphabet Σ the regular expressions are defined by the following BNF (Backus-Naur Form)

$$E ::= \emptyset \mid \epsilon \mid a \mid E + E \mid E^* \mid EE$$

This defines the *abstract syntax* of regular expressions to be contrasted with the *concrete syntax* (how we write regular expressions; see 3.1.3)

Concrete syntax

$01^* + 1$ means $(0(1^*)) + 1$

$(01)^* + 1$ is a different regular expression

$0(1^* + 1)$ yet another one

Regular Expressions: Abstract Syntax

Notice that

there is *no* intersection operation

there is *no* complement operation

Sometimes there are added (like in the Brzozozski algorithm that we shall explain later)

Regular expressions in functional programming

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

For instance

```
Plus (Atom "b") (Star (Concat (Atom "b") (Atom "c")))
```

is written $b + (bc)^*$.

Regular Expressions: Examples

If $\Sigma = \{a, b, c\}$

The expressions $(ab)^*$ represents the language

$\{\epsilon, ab, abab, ababab, \ldots\}$

The expression $(a + b)^*$ represents the words built only with a and b. The expression $a^* + b^*$ represents the set of strings with only as or with only bs (and ϵ is possible)

The expression $(aaa)^*$ represents the words built only with a, with a length divisible by 3

Regular Expressions: Examples

If $\Sigma = \{0,1\}$

$(\epsilon+1)(01)^*(\epsilon+0)$ is the set of strings that alternate 0's and 1's

Another expression for the same language is $(01)^*+1(01)^*+(01)^*0+1(01)^*0$

Some Operations on Languages

Three operations

1. *union* $L_1 \cup L_2$ of two languages L_1 and L_2

2. *concatenation* $L_1 L_2$ this is the set of all words $x_1 x_2$ with $x_i \in L_i$. If L_1 or L_2 is $\emptyset$ this is empty

3. *closure* L^* of a language; L^* is the union of ϵ and all words $x_1 \dots x_n$ with $x_i \in L$

Some Operations on Languages

Definition: $L^0 = \{\epsilon\}$, $L^{n+1} = L^n L$

Notice that $\emptyset^* = \{\epsilon\}$ and

$L^* = L^0 \cup L^1 \cup L^2 \cup \cdots = \bigcup_{n \in \mathbb{N}} L^n$

Semantics of regular expressions

This is defined by induction on the *abstract syntax*: $x \in L(E)$ iff x is *accepted* by E

1. $L(\emptyset) = \emptyset, \quad L(\epsilon) = \{\epsilon\}$

2. $L(a) = \{a\}$ if $a \in \Sigma$

3. $L(E_1 + E_2) = L(E_1) \cup L(E_2)$

4. $L(E_1 E_2) = L(E_1) L(E_2)$

5. $L(E^*) = L(E)^*$

Regular Languages and Regular Expressions

Theorem: *If L is a regular language there exists a regular expression E such that $L = L(E)$.*

We prove this in the following way.

To any automaton we associate a system of equations (the solution should be regular expressions)

We solve this system like we solve a *linear* equation system using *Arden's Lemma*

At the end we get a regular expression for the language recognised by the automaton. This works for DFA, NFA, ϵ-NFA

Regular Languages and Regular Expressions

For the automata with accepting states C and D and defined by

$$A.0 = \{A, B\},\ A.1 = B,\ B.0 = B.1 = C,\ C.0 = C.1 = D$$

We get the system

$$E_A = (0+1)E_A + 1E_B \qquad E_B = (0+1)E_C \qquad E_C = \epsilon + (0+1)E_D \qquad E_D = \epsilon$$

where $E_S = \{w \in \Sigma^* \mid S.w \cap F \neq \emptyset\}$

Arden's Lemma

Arden's Lemma: *A solution of* $x = Rx + S$ *is* $x = R^*S$*. Furthermore, if* $\epsilon \notin L(R)$ *then this is the* only *solution of the equation* $x = Rx + S$*.*

We have $R^* = RR^* + \epsilon$ and so $R^*S = RR^*S + S$

So $x = R^*S$ is a solution of $x = Rx + S$

Arden's Lemma

For the system

$$E_1 = bE_2 \qquad E_2 = aE_1 + bE_3 \qquad E_3 = \epsilon + bE_1$$

we get $E_1 = bE_2, \quad E_3 = \epsilon + bbE_2$ and then

$$E_2 = (ab + bbb)E_2 + b$$

and hence $E_2 = (ab + bbb)^*b$ and $E_1 = b(ab + bbb)^*b$

This is the same as the method described in 3.2.2 but it is expressed in the language of *equations* and *eliminating variables*

Regular Languages and Regular Expressions

We can find a solution of the original system by eliminating states

$$E_A = (0+1)E_A + 1E_B \qquad E_B = (0+1)E_C \qquad E_C = \epsilon + (0+1)E_D \qquad E_D = \epsilon$$

in the following way

$E_D = \epsilon,\ E_C = \epsilon + 0 + 1,\ E_B = 0 + 1 + (0+1)^2$ and

$$E_A = (0+1)^*(10 + 11 + 1(0+1)^2)$$

Regular Languages and Regular Expressions

How to remember the solution of $x = Rx + S$?

Notice that we have, if $x = Rx + S$?

$$x = Rx + S = R(Rx + S) + S = R^2x + RS + S$$

and so

$$x = R(R^2x + RS + S) + S = R^3x + R^2S + RS + S$$

and in general

$$x = R^{n+1}x + (R^n + \cdots + R + \epsilon)S$$

Regular Languages and Regular Expressions

The result depends on the way we solve the system

For $X = aX + bY,\ Y = \epsilon + cY + dX$

If we eliminate X first we get $X = a^*b(c + da^*b)^*$

If we eliminate Y first we get $X = (a + bc^*d)^*bc^*$

Hence $a^*b(c + da^*b)^* = (a + bc^*d)^*bc^*$!

Elimination of states

The books present two other methods.

The first method is similar to Warshall's algorithm (see wikipedia)

The second method is by elimination of states, and is in fact the same as the method of equations that I have presented (even if it does not look so similar at first)

Elimination of states

There is only one formula needed

$$E'_{ij} = E_{ij} + E_{ik}(E_{kk})^* E_{kj}$$

when we eliminate the state k

A nice trick (which is not in the book) is to add one extra initial state and one extra final state

Algorithm on regular expressions

Test if a regular expression denotes the empty language

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

isEmpty Empty = True
isEmpty (Plus e1 e2) = isEmpty e1 && isEmpty e2
isEmpty (Concat e1 e2) = isEmpty e1 || isEmpty e2
isEmpty _ = False
```

Algorithm on regular expressions

Test if a regular expression contains ϵ

```
hasEpsilon :: Reg 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
```

Algorithm on regular expressions

Test if $L(e) \subseteq \{\epsilon\}$

```
atMostEps :: Reg a -> Bool

atMostEps Empty = True
atMostEps Epsilon = True
atMostEps (Star e) = atMostEps e
atMostEps (Plus e1 e2) = atMostEps e1 && atMostEps e2
atMostEps (Concat e1 e2) =
 Empty e1 || Empty e2 || (atMostEps e1 && atMostEps e2)
atMostEps  _ = False
```

Algorithm on regular expressions

Test if a regular expression denotes an *infinite* language

```
infinite :: Reg a -> Bool

infinite (Star e) = not (atMostExp e)
infinite (Plus e1 e2) = infinite e1 || infinite e2
infinite (Concat e1 e2) =
 (infinite e1 && notEmpty e2) || (notEmpty e1 && infinite e2)
infinite  _ = False

notEmpty e = not (isEmpty e)
```

Derivative of a regular expression

If L is a language $L \subseteq \Sigma^*$ and $a \in \Sigma$ we define the language L/a (derivative of L by a) by

$$L/a = \{x \in \Sigma^* \mid ax \in L\}$$

We give an algorithm computing E/a such that $L(E/a) = L(E)/a$ using the equivalence

$ax \in L$ iff $x \in L/a$

Derivative of a regular expression

Examples

$(abab + abba)/a = bab + bba$

$(abab + abba)/b = \emptyset$

$(a^*b)/a = (aa^*b + b)/a = a^*b$

$((ab)^*a)/a = (ab(ab)^*a + a)/a = b(ab)^*a + \epsilon$

Derivative of a regular expression

```
der :: Eq a => a -> Reg a -> Reg a

der b (Atom b1) = if b == b1 then Epsilon else Empty
der b (Plus e1 e2) = Plus (der b e1) (der b e2)
der b (Concat e1 e2) | hasEpsilon e1 =
 Plus (Concat (der b e1) e2) (der b e2)
der b (Concat e1 e2) = Concat (der b e1) e2
der b (Star e) = Concat (der b e) (Star e)
der b _ = Empty
```

Application

Is a given word in the language defined by a regular expression E?

```
isIn :: Eq a => [a] -> Reg a -> Bool

isIn [] e = hasEpsilon e
isIn (a:as) e = isIn as (der a e)
```

This is essentially Ken Thompson's algorithm

This works if we add *intersection* and *complement*

Application: extended regular expressions

```
data Reg a =
 Empty | Epsilon | Atom a | Plus (Reg a) (Reg a) |
 Concat (Reg a) (Reg a) | Star (Reg a) |
 Inter (Reg a) (Reg a) | Compl (Reg a)

hasEpsilon Epsilon = True
hasEpsilon (Star _) = True
hasEpsilon (Inter e1 e2) = hasEpsilon e1 && hasEpsilon e2
hasEpsilon (Compl e) = not (hasEpsilon e)
hasEpsilon (Plus e1 e2) = hasEpsilon e1 || hasEpsilon e2
hasEpsilon (Concat e1 e2) = hasEpsilon e1 && hasEpsilon e2
hasEpsilon  _ = False
```

Application: extended regular expressions

```
der :: Eq a => a -> Reg a -> Reg a

der b (Atom b1) = if b == b1 then Epsilon else Empty
der b (Plus e1 e2) = Plus (der b e1) (der b e2)
der b (Inter e1 e2) = Inter (der b e1) (der b e2)
der b (Compl e) = Compl (der b e)
der b (Concat e1 e2) | hasEpsilon e1 =
 Plus (Concat (der b e1) e2) (der b e2)
der b (Concat e1 e2) = Concat (der b e1) e2
der b (Star e) = Concat (der b e) (Star e)
der b _ = Empty
```

Application

Is a given word in the language defined by a regular expression E? The algorithm is the same

```
isIn :: Eq a => [a] -> Reg -> Bool

isIn [] e = hasEpsilon e
isIn (a:as) e = isIn as (der a e)
```

Derivatives

Example: $x = abba$ and $E = abba + abab$

The algorithm works with generalised regular expressions

$x = 1010$ and $E = (01 + 10)^* \cap (101)^*$

Regular Languages and Regular Expressions

Theorem: *If E is a regular expression then $L(E)$ is a regular language*

We prove this by induction on E. The main steps are to prove that

if L_1, L_2 are regular then so is $L_1 \cup L_2$ and $L_1 L_2$

if L is regular then so is L^*

Regular Languages and Regular Expressions

At the end we shall get an ϵ-*NFA* that we know how to transform into a DFA by the subset construction

There is a beautiful algorithm that builds directly a DFA from a regular expression, due to Brzozozski, and we present also this algorithm

Regular Languages and Regular Expressions

Lemma: *If L_1, L_2 are regular then so is $L_1 \cup L_2$*

We have seen a proof of this with the product construction. This is easy also if $L_1 = L(A_1),\ L_2 = L(A_2)$ and A_1, A_2 are ϵ-NFAs

Lemma: *If L_1, L_2 are regular then so is $L_1 L_2$*

Lemma: *If L is regular then so is L^**

Regular Languages and Regular Expressions

This can be seen as an algorithm transforming a regular expression E to an ϵ-NFA

Example: we transform $a^* + ab$ to an ϵ-NFA

As you can see on this example the automaton we obtain is quite complex

A priori even more complex if we want a DFA

See also Figure 3.18

Brzozozski's algorithm

The idea is to use *derivatives* as *states*

For instance if $E = a^* + ab$ we have

$E/a = a^* + b,\ E/b = \emptyset$

$E/aa = a^*,\ E/ab = \epsilon,\ E/ba = E/bb = E/b$

$E/aaa = E/aa,\ E/aab = E/aba = E/abb = E/b$

We get a DFA with 5 states. The accepting states are the ones that contain ϵ

Brzozozski's algorithm

Other examples

$E = (a + \epsilon)^*$

$E = F10F$ where $F = (0 + 1)^*$

$E = F1(0 + 1)$

Brzozozski's algorithm

Furthermore this algorithm works even with *extended* regular expressions that admit intersections and complements

For instance $E = a(ba)^* - (ab)^*a$

$E/a = (ba)^* - (b(ab)^*a + \epsilon),\ \ E/b = \emptyset$

$E/aa = \emptyset,\ \ E/ab = a(ab)^* - (ab)^*a = E$

and none of these expressions contains ϵ, so $E = \emptyset$!

Brzozozski's algorithm

Example: We can prove in this way

$(01+10)^* \cap (101)^* = \epsilon$

More generally we get an algorithm for testing $E = F$: we build a tree with nodes $E/x, F/x$ for finite x

Examples: $E = (01+10)^*,\ F = (101)^*$

$E = (10)^*1,\ F = 1(01)^*$

Algebraic Laws for Languages

$L_1 \cup L_2 = L_2 \cup L_1$ Union is *commutative*

Note: Concatenation is *not* commutative we can find L_1, L_2 such that $L_1L_2 \neq L_2L_1$

$L\{\epsilon\} = \{\epsilon\}L = L$

$L\emptyset = \emptyset L = \emptyset$

$L(M \cup N) = LM \cup LN$

$(M \cup N)L = ML \cup NL$

Algebraic Laws for Languages

$\emptyset^* = \{\epsilon\}^* = \{\epsilon\}$

$L^+ = LL^* = L^*L$

$L? = L \cup \{\epsilon\}$

$(L^*)^* = L^*$

Algebraic Laws for Regular Expressions

We write $E = F$ for $L(E) = L(F)$

For instance

$$(E_1 + E_2)E = E_1E + E_2E$$

follows from

$$(L_1 \cup L_2)L = L_1L \cup L_2L$$

by taking $L_i = L(E_i),\ L = L(E)$

Similarly $(E^*)^* = E^*$

Algebraic Laws for Regular Expressions

$$E + (F + G) = (E + F) + G,\ E + F = F + E,\ E + E = E,\ E + 0 = E$$

$$E(FG) = (EF)G,\ E0 = 0E = 0,\ E\epsilon = \epsilon E = E$$

$$E(F + G) = EF + EG,\ (F + G)E = FE + GE$$

$$\epsilon + EE^* = E^* = \epsilon + E^*E$$

Algebraic Laws for Regular Expressions

We have also

$E^* = E^*E^* = (E^*)^*$

$E^* = (EE)^* + E(EE)^*$

Algebraic Laws for Regular Expressions

How can one prove equalities between regular expressions?

In usual algebra, we can "simplify" an algebraic expression by rewriting

$$(x+y)(x+z) \rightarrow xx + yx + xz + yz$$

For regular expressions, there is no such way to prove equalities. There is not even a complete finite set of equations.

Algebraic Laws for Regular Expressions

Example: $L^* \subseteq L^*L^*$ since $\epsilon \in L^*$

Conversely if $x \in L^*L^*$ then $x = x_1x_2$ with $x_1 \in L^*$ and $x_2 \in L^*$

$x \in L^*$ is clear if $x_1 = \epsilon$ or $x_2 = \epsilon$. Otherwise

So $x_1 = u_1 \ldots u_n$ with $u_i \in L$

and $x_2 = v_1 \ldots v_m$ with $v_j \in L$

Then $x = x_1x_2 = u_1 \ldots u_n v_1 \ldots v_m$ is in L^*

Algebraic Laws for Regular Expressions

Two laws that are useful to simplify regular expressions

Shifting rule

$$E(FE)^* = (EF)^*E$$

Denesting rule

$$(E^*F)^*E^* = (E+F)^*$$

Variation of the denesting rule

One has also

$$(E^*F)^* = \epsilon + (E+F)^*F$$

and this represents the words empty or finishing with F

Algebraic Laws for Regular Expressions

Example:

$a^*b(c + da^*b)^* = a^*b(c^*da^*b)^*c^*$

by denesting

$a^*b(c^*da^*b)^*c^* = (a^*bc^*d)^*a^*bc^*$

by shifting

$(a^*bc^*d)^*a^*bc^* = (a + bc^*d)^*bc^*$

by denesting. Hence

$a^*b(c + da^*b)^* = (a + bc^*d)^*bc^*$

Algebraic Laws for Regular Expressions

Examples: $10?0? = 1 + 10 + 100$

$$(1 + 01 + 001)^*(\epsilon + 0 + 00) = ((\epsilon + 0)(\epsilon + 0)1)^*(\epsilon + 0)(\epsilon + 0)$$

is the same as

$$(\epsilon + 0)(\epsilon + 0)(1(\epsilon + 0)(\epsilon + 0))^* = (\epsilon + 0 + 00)(1 + 10 + 100)^*$$

Set of all words with no substring of more than two adjacent 0's

Proving by induction

Let Σ be $\{a, b\}$

Lemma: *For all* n *we have* $a(ba)^n = (ab)^n a$

Proof: by induction on n

Theorem: $a(ba)^* = (ab)^* a$

Similarly we can prove $(a + b)^* = (a^* b)^* a^*$

Complement of a(n ordinary) regular expression

For building the "complement" of a regular expression, or the "intersection" of two regular expressions, we can use NFA/DFA

For instance to build E such that $L(E) = \{0,1\}^* - \{0\}$ we first build a DFA for the expression 0, then the complement DFA. We can compute E from this complement DFA. We get for instance

$$\epsilon + 1(0+1)^* + 0(0+1)^+$$