# Finite automata and regular expressions in Haskell

N. Batistoni, M. Lürsen, P. Fink, J. Sorkin

Sunday 26th May, 2024

**Abstract**

We implement the data types for deterministic and non-deterministic finite automata, as well as for regular expressions. Moreover, we implement the proofs of their equivalence in describing regular languages from Chapter 1 of [Sip12].

Notes for the peer reviewers:

- to compile everything, use `stack build`;
- to play with the code in ghci, use `stack ghci`;
- there is an executable, but it is still the one from the template which doesn't really do anything;
- to run the tests from § 4, use `stack clean && stack test`.

## Contents

# 1 DFAs and NFAs

**Definition 1.** We define a deterministic finite automaton (DFA) as a 5-tuple $\langle Q, \Sigma, \delta, q_0, F\rangle$ where

(i) Q is a finite set of states,

(ii) $\Sigma$ is a finite set of symbols (the alphabet),

(iii) $\delta^{DFA} : Q \times \Sigma \to Q$ is a transition function,

(iv) $q_0 \in Q$ is the start state,

(v) $F \subseteq Q$ is a set of final states.

**Definition 2.** We define a nondeterministic finite automaton (NFA) as a 5-tuple $\langle Q, \Sigma, \delta, q_0, F\rangle$ where

(i) Q is a finite set of states,

(ii) $\Sigma$ is a finite set of symbols (the alphabet),

(iii) $\delta^{NFA} : Q \times \Sigma \cup \{\varepsilon\} \to \mathcal{P}(Q)$ is a transition function,

(iv) $q_0 \in Q$ is the start state,

(v) $F \subseteq Q$ is a set of final states.

We have implemented these definitions as closely as possible in the data type definitions below. There are a couple of things to note about this. First, notice how the $\delta^{DFA}$ function maps a tuple of type `state` and `symbol` to the type `Maybe state`. The reason for this is that $\delta^{DFA}$ can be a partial function, potentially leading to exceptions when excecuting functions call the transition function. To handle such exceptions more easily we implement $\delta^{DFA}$ to map to `Maybe state`, returning `Nothing` whenever the function is not defined for a particular combination of $(st, sy)$. We make the necessary steps to and from the `Maybe` context within the functions requiring such conversions themselves. Second, $\delta^{NFA}$ maps a tuple of type `state` and `Maybe symbol` to the type `[state]`. We choose to represent $\Sigma \cup \{\varepsilon\}$ using `Maybe symbol` as it provides the additional value to the alphabet by which we can represent $\varepsilon$-transitions. Here too we make the conversion to and from `Maybe` within the functions that require these conversions themselves.

```haskell
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE RankNTypes #-}
module DfaAndNfa where

-- import Test.QuickCheck
import Data.Maybe ( fromMaybe )

data DFA state symbol = DFA
                    { statesDFA :: [state]
                    , alphabetDFA :: [symbol]
                                                     -- Nothing : state is our "garbage"
                                                        state.
                                                     -- Reject all strings the moment
                                                       they reach Nothing : state.
                    , transitionDFA :: (state,symbol) -> Maybe state
                    , beginDFA :: state
                    , finalDFA :: [state]
                    }


data NFA state symbol = NFA
                    { statesNFA :: [state]
                    , alphabetNFA :: [symbol]
                    , transitionNFA :: (state, Maybe symbol) -> [state]
                    , beginNFA :: state
                    , finalNFA :: [state]
                    }

-- Dummy DFA for testing purposes
testDFA :: DFA Integer Char
testDFA = DFA   [1,2]
                "ab"
                (`lookup` [((1,'a'), 1), ((1,'b'), 2)])
                1
                [2]

-- Dummy NFA for testing purposes
testNFA :: NFA Integer Char
testNFA = NFA   [1,2,3]
                "ab"
                (\(st,sy) -> fromMaybe [] $ lookup (st,sy)
                    [ ((1, Just 'a'), [1]), ((1, Just 'b'), [1,2])
                    , ((1, Nothing), [2]), ((2, Just 'a'), [2])
                    , ((2,Just 'b'), [2]), ((2, Nothing), [3])
                    , ((3, Just 'a'), [2]), ((3, Nothing), [1])]
                )
                1
                [2]

-- evaluate function for DFA. Checks whether a given string of symbols results in a final
    state.
evaluateDFA :: forall state symbol . Eq state => DFA state symbol -> [symbol] -> Bool
evaluateDFA (DFA _ _ delta begin final) syms = case walkDFA (Just begin) syms of
    Nothing -> False
    Just s -> s `elem` final
    where -- ugly helper function to handle the Maybe's
        walkDFA :: Maybe state -> [symbol] -> Maybe state
        walkDFA Nothing _ = Nothing
        walkDFA (Just q) [] = Just q
        walkDFA (Just q) (s:ss) = case delta (q,s) of
            Nothing -> Nothing
            Just q' -> walkDFA (Just q') ss

-- Close the set {x} under epsilon-arrows
epsilonClosure :: forall state symbol . Eq state => NFA state symbol -> state -> [state]
epsilonClosure nfa x = closing [] [x] where
    closing visited [] = visited
    closing visited (y:ys)
        | y `elem` visited = closing visited ys
        | otherwise = closing (y : visited) (ys ++ transitionNFA nfa (y, Nothing))

-- This is U_{x in xs} epsilonClosure nfa x
epsilonClosureSet :: Eq state => NFA state symbol -> [state] -> [state]
epsilonClosureSet nfa = concatMap (epsilonClosure nfa)

-- Implementation from here: https://en.wikipedia.org/wiki/
    Nondeterministic_finite_automaton
evaluateNFA :: forall state symbol . Eq state => NFA state symbol -> [symbol] -> Bool
evaluateNFA nfa syms = any (`elem` finalNFA nfa) (walkNFA (beginNFA nfa) (reverse syms))
    where
    walkNFA :: state -> [symbol] -> [state]
    -- delta*(q, epsilon) = E {q}
    walkNFA   q  []        = epsilonClosureSet nfa [q]
    -- delta*(q, w ++ [a]) = U_{r in delta*(q, w)} E(delta(r,a))
    walkNFA q (a : w)      = concatMap (\r -> epsilonClosureSet nfa (delta (r, Just a)))
        walkNFA' where
        delta = transitionNFA nfa
        -- delta*(q, w)
        walkNFA' = walkNFA q w

-- Secondary evaluate function for NFAs. There is some error in one of the two, but we have
     yet to find out which.
evaluateNFA' :: forall state symbol . Eq state => NFA state symbol -> [symbol] -> Bool
evaluateNFA' nfa syms = any (`elem` finalNFA nfa) (walkNFA [beginNFA nfa] syms) where
    walkNFA :: [state] -> [symbol] -> [state]
    walkNFA states [] = epsilonClosureSet nfa states
    walkNFA states (s:ss) = walkNFA (concatMap transition epsilonClosureStates) ss where
        transition q = transitionNFA nfa (q, Just s)
        epsilonClosureStates = epsilonClosureSet nfa states


printDFA :: (Show state, Show symbol) => DFA state symbol -> String
printDFA (DFA states alphabet transition begin final) =
    "States: " ++ show states ++ "\n" ++
    "Alphabet: " ++ show alphabet ++ "\n" ++
    "Start State: " ++ show begin ++ "\n" ++
    "Final States: " ++ show final ++ "\n" ++
    "Transitions:\n" ++ unlines (map showTransition allTransitions)
  where
    showTransition ((state, sym), nextState) =
        show state ++ " -- " ++ show sym ++ " --> " ++ show nextState
    allTransitions = [((state, sym), transition (state, sym)) | state <- states, sym <-
        alphabet ]

printNFA :: (Show state, Show symbol) => NFA state symbol -> String
printNFA (NFA states alphabet transition begin final) =
    "States: " ++ show states ++ "\n" ++
    "Alphabet: " ++ show alphabet ++ "\n" ++
    "Start State: " ++ show begin ++ "\n" ++
    "Final States: " ++ show final ++ "\n" ++
    "Transitions: \n" ++ unlines (map showTransition allTransitions)
  where
    showTransition ((state, Nothing), nextStates) =
        show state ++ " -- " ++ "eps" ++ " --> " ++ show nextStates
    showTransition ((state, Just sym), nextStates) =
        show state ++ " -- " ++ show sym ++ " --> " ++ show nextStates
    allTransitions = [((state, sym), transition (state, sym)) | state <- states, sym <-
        Nothing : map Just alphabet, not $ null $ transition (state,sym)]

{-
--TODO:
-- Arbitrary instance for DFA. This is necessary for implementing the autotests.
instance Arbitrary DFA state symbol where
arbitrary :: Gen DFA state symbol
arbitrary = undefined

-- Arbitrary instance for NFA. This is necessary for implementing the autotests.
instance Arbitrary NFA state symbol where
arbitrary :: Gen NFA state symbol
arbitrary = undefined

-- Show instance for DFA
instance (Show state, Show symbol) => Show (DFA state symbol) where
    show :: (Show state, Show symbol) => DFA state symbol -> String
    show dfa = "DFA {\n" ++
               "  statesDFA = " ++ show (statesDFA dfa) ++ ",\n" ++
               "  alphabetDFA = " ++ show (alphabetDFA dfa) ++ ",\n" ++
               "  transitionDFA = fromJust . flip lookup " ++ show (transitionListDFA dfa)
                ++ ",\n" ++
               "  beginDFA = " ++ show (beginDFA dfa) ++ ",\n" ++
               "  finalDFA = " ++ show (finalDFA dfa) ++ "\n" ++
               "}"
                where
                    -- Generates lookup table
                    transitionListDFA :: DFA state symbol -> [((state,symbol),state)]
                    transitionListDFA = undefined

-- Show instance for NFA
instance (Show state, Show symbol) => Show (NFA state symbol) where
    show :: (Show state, Show symbol) => NFA state symbol -> String
    show nfa = "NFA {"++
               "  statesNFA = " ++ show (statesNFA nfa) ++ ",\n" ++
               "  alphabetNFA = " ++ show (alphabetNFA nfa) ++ ",\n" ++
               "  transitionNFA = fromMaybe [] $ lookup " ++ show (transitionListNFA nfa)
                ++
               "  beginNFA = " ++ show (beginNFA nfa) ++
               "  finalNFA = " ++ show (finalNFA nfa) ++
               "}"
               where
                    -- Generates lookup table
                    transitionListNFA :: NFA state symbol -> [((state,symbol), [state])]
                    transitionListNFA = undefined
-}
```

## 1.1 The Powerset construction

In this section, we implement the Powerset construction. The powerset construction is an algorithm that transforms a NFA into a equivalent DFA where equivalent means that they accept exactly the same strings.

```
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE ScopedTypeVariables #-}
module NfaToDfa where

import DfaAndNfa
    ( DFA(DFA, beginDFA, transitionDFA, finalDFA, alphabetDFA),
      NFA(NFA, transitionNFA),
      epsilonClosure )
import Data.Maybe ( fromMaybe, mapMaybe )
import Data.List ( intersect, nub )
```

We straight forwardly implement the powersetconstruction . Here, we translate a NFA, $N = (Q, \Sigma, T, q_0, F)$, where $Q$ is the set of states, $\Sigma$ is the alphabet, $T$ is the transitions function $T : Q \times \Sigma \to Q$, $q_0 \in Q$ the initial state and $F \subseteq Q$ the set of final states. We define the corrsponding DFA as $D = (Q', \Sigma, T', q, F')$ where

- $Q' = \mathcal{P}(Q)$
- $T' : Q' \times \Sigma \to Q', T'((S, x)) = T'(\bigcup_{q \in S}\{T(q, x)\}, \epsilon)$
- $q = T(q_0, \epsilon)$
- $F' = Q' \cap F$

```

powerSetList :: [a] -> [[a]]
powerSetList [] = [[]]
powerSetList (x:xs) = map (x:) (powerSetList xs) ++ powerSetList xs

nfaToDfa :: Eq state => NFA state symbol -> DFA [state] symbol
nfaToDfa (NFA statesN alphabetN transN startN endN) =
  let nfa = NFA statesN alphabetN transN startN endN
      statesD = powerSetList statesN                                      -- new set of
           states
      alphabetD = alphabetN                                               -- same
          alphabet as the NFA
      startD = epsilonClosure nfa startN                                  -- the set of
           all states reachable from initial states in the NFA by epsilon-moves
      endD = filter (\state -> not $ null (state `intersect` endN)) statesD   -- All states
           that contain an endstate.
      transD (st, sy) =                                                   --
```

```
          Just $ nub $ concatMap (epsilonClosure nfa) syTransitionsForDfaStates where
                    -- epsilonClosure of the sy-reachable states
            syTransitionsForDfaStates = concatMap (\s -> transitionNFA nfa (s, Just sy)) st
                  -- states reachable by sy-transitions
  in  DFA statesD alphabetD transD startD endD
```

To minimize the DFA, we first find all the unreachable states and then delete them in the next step. To find all the unreachable states, we start from the initial state and then check whether there is a string that allows one to reach that state from the initial state. The "nextStates" function, takes a state and returns all states reachable by any character in the alphabet. We use this "nextStates" in the "closing" function. This function takes two lists of states as arguments and returns another list of states. The returned list contains all states that can be reached from the second list. To not end up in loops, we keep track of all states already visited using a list "visited".

We use the function "findReachableStates" to define the set of states in the new DFA which are just all states that are reachable from the initials state. Then, we restrict the transitions and final states to the reachable states in the original DFA.

```
findReachableStatesDFA :: forall state symbol . Eq state => DFA state symbol -> [state] ->
    [state]
findReachableStatesDFA dfa initialStates = nub $ closing [] initialStates where
  closing :: Eq state => [state] -> [state] -> [state]
  closing visited [] = visited
  closing visited (y:ys)
    | y `elem` visited = closing visited ys
    | otherwise = closing (y : visited) (ys ++ nextStates y)
  nextStates :: state -> [state]
  nextStates state = mapMaybe (\sym -> transitionDFA dfa (state, sym)) (alphabetDFA dfa) --
       checks for the next states following "state" for any symbol

-- Function to remove unreachable states from a DFA

removeUnreachableStates :: (Eq state, Eq symbol) => DFA state symbol -> DFA state symbol
removeUnreachableStates dfa = DFA reachableStates (alphabetDFA dfa) newTransition (beginDFA
     dfa) newFinalStates where
  reachableStates = findReachableStatesDFA dfa [beginDFA dfa] -- Other states cannot play a
       role in the evaluation of strings
  transitionsToReachables = [((s, a), transitionDFA dfa (s, a)) | s <- reachableStates, a
      <- alphabetDFA dfa]
  newTransition (s, a) = fromMaybe (error "Invalid transition") (lookup (s, a)
      transitionsToReachables)
  newFinalStates = filter (`elem` reachableStates) (finalDFA dfa)
```

## 2 Regular Expressions

**Definition 3.** Fix an alphabet $\Sigma$. We say that $R$ is *regular expression* over $\Sigma$ if:

(i) $R = a$ for some $a \in \Sigma$;

(ii) $R = \varnothing$,

(iii) $R = \varepsilon$,

(iv) $R = R_1 \cup R_2$, where $R_1, R_2$ are regular expressions,

(v) $R = R_1 \cdot R_2$, where $R_1, R_2$ are regular expressions,

(vi) $R = R_1^*$, where $R_1$ is a regular expression.

It is also often useful to use the abbreviation $R^+ := R \cup R^*$.

The following data type definition implements the `RegExp` type by closely following its formal definition. Together with the binary union (`Or`) and concatenation (`Concat`) operators, we also define their $n$-ary versions for convenience, as well as the `oneOrMore` abbreviation for $+$. Finally, we implement a function for displaying regular expressions in a more readable format[1].

```haskell
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE InstanceSigs #-}

module RegExp where

import Test.QuickCheck ( Arbitrary(arbitrary), Gen, oneof, sized )

data RegExp sym = Empty
                | Epsilon
                | Literal sym
                | Or (RegExp sym) (RegExp sym)
                | Concat (RegExp sym) (RegExp sym)
                | Star (RegExp sym)
                deriving (Eq,Show)

oneOrMore :: RegExp sym -> RegExp sym
oneOrMore re = re `Concat` Star re

orAll :: [RegExp sym] -> RegExp sym
orAll = foldr Or Empty

concatAll :: [RegExp sym] -> RegExp sym
concatAll = foldr Concat Epsilon

prettyPrint :: Show sym => RegExp sym -> String
prettyPrint re = case re of
    Empty -> "\2205"                                      -- unicode for \varnothing
    Epsilon -> "\0949"                                    -- unicode for \varepsilon
    Literal l -> show l
    Or re1 re2 -> "(" ++ prettyPrint re1 ++ "|" ++ prettyPrint re2 ++ ")"
    Concat re1 re2 -> prettyPrint re1 ++ prettyPrint re2
    Star re1 -> "(" ++ prettyPrint re1 ++ ")*"
```

Formally, the language described by a regular expression $R$ over $\Sigma$ is denoted $L(R)$ and consists exactly of the strings over $\Sigma$ that match $R$: intuitively, these are the strings that match the pattern specified by $R$, where all operators are interpreted in the obvious way, and the $*$ stands for “arbitrary number of repetitions of the pattern”.

**Definition 4.** Let $R$ be a regular expression and $s$ a string, over the same alphabet $\Sigma$. We say that $s$ matches $R$ if:

(i) if $R = \varnothing$, then never;

(ii) if $R = \varepsilon$ and $s = \varepsilon$;

(iii) if $R = a \in \Sigma$ and $s = a$;

[1]This technically operates under the assumption that the alphabet does not contain `*` or `+` or the parentheses symbols, which would make the `prettyPrint` output ambiguous. Since the only purpose of this function is to display regular expressions in a readable format, however, we choose to simply ignore the issue.

(iv) if $R = R_1 \cup R_2$, and $s$ matches $R_1$ or $s$ matches $R_2$;

(v) if $R = R_1 \cdot R_2$, and there exist $s_1, s_2$ such that $s = s_1 s_2$ and $s_1$ matches $R_1$ and $s_2$ matches $R_2$;

(vi) if $R = R_1^*$, and $s = \varepsilon$ or $s$ can be split into $n \in \mathbb{N}$ substrings $s_1, \ldots, s_n$ such that every $s_i$ matches $R_1$.

The following function implements matching in a straightforward way. In our tests, it will essentially play the same role as the `evaluateDFA` and `evaluateNFA` functions, and we will use it to check whether (supposedly) equivalent automata and regular expressions do accept/match the same strings.

```
matches :: Eq sym => [sym] -> RegExp sym -> Bool
matches str re = case re of
    Empty -> False
    Epsilon -> null str
    Literal l -> str == [l]
    Or re1 re2 -> matches str re1 || matches str re2
    Concat re1 re2 -> or [ matches str1 re1 && matches str2 re2 | (str1, str2) <-
        allSplittings str ] where
        allSplittings s = [ splitAt k s | k <- [0..n] ] where n = length s
    Star re1 -> matches str Epsilon || or [ matches str1 re1 && matches str2 (Star re1) | (
        str1, str2) <- allNonEmptySplittings str ] where
        allNonEmptySplittings s = [ splitAt k s | k <- [1..n] ] where n = length s
```

Next, we implement a function to simplify regular expressions using some algebraic identities[2]. Note that this function does not minimize a given regular expression[3] but it is useful in improving its readability, especially for the regular expressions that we will obtain by converting NFAs. Moreover, since the conversions are very inefficient and result in very large regular expressions, simplifying them will help speed up the tests.

```
simplify :: Eq sym => RegExp sym -> RegExp sym
simplify re -- repeatedly apply the one-step simplify function until a fixed point is
    reached
    | oneStepSimplify re == re = re
    | otherwise = simplify $ oneStepSimplify re
    where
        oneStepSimplify :: Eq sym => RegExp sym -> RegExp sym
        oneStepSimplify Empty = Empty
        oneStepSimplify Epsilon = Epsilon
        oneStepSimplify (Literal l) = Literal l
        oneStepSimplify (Or re1 re2)
            | re1 == Empty = oneStepSimplify re2
            | re2 == Empty = oneStepSimplify re1
            | re1 == re2 = oneStepSimplify re1
            | otherwise = Or (oneStepSimplify re1) (oneStepSimplify re2)
        oneStepSimplify (Concat re1 re2)
            | re1 == Empty || re2 == Empty = Empty
            | re1 == Epsilon = oneStepSimplify re2
            | re2 == Epsilon = oneStepSimplify re1
            | otherwise = Concat (oneStepSimplify re1) (oneStepSimplify re2)
        oneStepSimplify (Star re') = case re' of
            Empty -> Epsilon
            Epsilon -> Epsilon
            Or Epsilon re2 -> Star (oneStepSimplify re2)
            Or re1 Epsilon -> Star (oneStepSimplify re1)
            Star re1 -> Star (oneStepSimplify re1)
            _ -> Star (oneStepSimplify re')
```

[2]Which we will state here for the final version of the report.
[3]This is a very hard computational problem, see e.g. ...

Finally, we implement a way to generate random regular expressions using QuickCheck. We try to keep their size relatively small so that the NFA to regular expression conversion does not take too long.

```haskell
instance Arbitrary sym => Arbitrary (RegExp sym) where
  arbitrary :: Arbitrary sym => Gen (RegExp sym)
  arbitrary = sized randomRegExp where
    randomRegExp :: Int -> Gen (RegExp sym)
    randomRegExp 0 = oneof [ Literal <$> (arbitrary :: Gen sym), return Epsilon, return
        Empty ]
    randomRegExp n = oneof [ Literal <$> (arbitrary :: Gen sym), return Epsilon
                           , Or <$> randomRegExp (n `div` 10) <*> randomRegExp (n `div` 10)
                           , Concat <$> randomRegExp (n `div` 10) <*> randomRegExp (n `div`
                             10)
                           , Star <$> randomRegExp (n `div` 10)
                           ]
```

## 3 Equivalence of finite automata and regular expressions

In this section, our goal is to implement the constructive proof of Theorem 1.54 from [Sip12].

**Theorem 5.** *A language is regular if and only if it is described by a regular expression.*

In § 3.1, we implement the construction of an NFA from a regular expression to show that if a language is described by a regular expression, then it is regular. Next, in § 3.2, we implement the construction of a regular expression from a given NFA to prove that if a language is regular, then it is described by a regular expression.

### 3.1 Converting regular expressions to NFAs

Here, we state and implement the proof of the following lemma. Since the implementation is very straightforward, we first prove the lemma and then briefly discuss a few notable implementation details.

**Lemma 6.** *If a language is described by a regular expression, then it is regular.*

*Proof.* Fix an arbitrary alphabet $\Sigma$ and let $R$ be a regular expression over $\Sigma$. The proof is by induction on the structure of $R$[4].

Case $R = \varnothing$. Then $L(R) = \varnothing$ is accepted by the NFA $(\{q_0\}, \Sigma, \delta, q_0, \varnothing)$ where $\delta(q, s) = \varnothing$ for every $q \in Q$ and $s \in \Sigma$.

Case $R = \varepsilon$. Then $L(R) = \{\varepsilon\}$ is accepted by the NFA $(\{q_0\}, \Sigma, \delta, q_0, \{q_0\})$ where $\delta(q, s) = \varnothing$ for every $q \in Q$ and $s \in \Sigma$.

Case $R = \ell \in \Sigma$. Then $L(R) = \{\ell\}$ is accepted by the NFA $(\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$ where $\delta(q_0, \ell) = \{q_1\}$ and $\delta(q, s) = \varnothing$ otherwise.

[4]Might add some pictures?

Case $R = R_1 \cdot R_2$. By the inductive hypothesis, there are NFAs $N_1$ and $N_2$ accepting $L(R_1)$ and $L(R_2)$ respectively. We can construct an NFA $N$ that accepts $L(R)$ by adding epsilon-transitions from $N_1$'s final states to $N_2$'s start state, "guessing" where to break the input so that $N_1$ accepts its first substring and $N_2$ its second. Formally, let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$. Then we can define $N = (Q, \Sigma, \delta, q_1, F_2)$, where $Q = Q_1 \cup Q_2$, and

$$\delta(q, s) = \begin{cases} \delta_1(q, s) \text{ if } q \in Q_1 \setminus F_1; \\ \delta_1(q, s) \text{ if } q \in F_1 \text{ and } s \neq \varepsilon; \\ \delta_1(q, s) \cup \{q_2\} \text{ if } q \in F_1 \text{ and } s = \varepsilon; \\ \delta_2(q, s) \text{ if } q \in Q_2. \end{cases}$$

It is then clear that $L(N) = L(R_1 \cdot R_2)$.

Case $R = R_1 \cup R_2$. The idea is to glue the NFAs $N_1$ and $N_2$ given by the induction hypothesis to a new start state which has epsilon-transitions to the start states of $N_1$ and $N_2$, so as to "guess" whether the input string is in $L(R_1)$ or $L(R_2)$. (Will be further explained)

Case $R = (R_1)^*$. We add a new start and final state to the NFA $N_1$ given by the induction hypothesis, with an epsilon-transition from this state to $N_1$'s start state. Moreover, we add epsilon-transitions from $N_1$'s final states to $N_1$'s start state. (Will be further explained) □

The implementation of the construction described in the proof is very straightforward, with only a couple implementation details. First, since we do not have a way to know which specific alphabet a regular expression is defined over, we have to manually define or augment the alphabets in each case. The definition of the new transition functions slightly changes accordingly. Moreover, we need a way to keep track of which labels have been used for the NFA's states. An easy way to do this is generating an NFA whose states are labelled as `Int`. To keep track of the last used `Int`, we use an auxiliary function `regexToNfaHelper` to actually construct the NFAs. The function takes an `Int` parameter representing the first available integer to label the states, and return an `NFA,Int` pair which includes the next available integer. In short, `regexToNfaHelper` does all the work, and the outer `regexToNfa` function simply returns the so-constructed NFA discarding the `Int` output.

```haskell
module RegToNfa where

import RegExp ( RegExp(..) )
import DfaAndNfa ( NFA(NFA) )
import Data.List ( union )
import Data.Maybe ( isNothing )

regexToNfa :: Eq sym => RegExp sym -> NFA Int sym
regexToNfa re = fst $ regexToNfaHelper re 1 where
    -- auxiliary function used to build an NFA equivalent to the given regex
    -- its second parameter is the first available int to name the NFA's states
    -- returns the NFA built from the smaller regex's, and the next first available int
    regexToNfaHelper :: Eq sym => RegExp sym -> Int -> (NFA Int sym, Int)
    regexToNfaHelper Empty n = ( NFA [n] [] delta n [], n+1 ) where delta (_,_) = []
    regexToNfaHelper Epsilon n = ( NFA [n] [] delta n [n], n+1 ) where delta (_,_) = []
    regexToNfaHelper (Literal l) n = ( NFA [n,n+1] [l] delta n [n+1], n+2 ) where
        delta (st,sy)
            | st == n && sy == Just l = [n+1]
            | otherwise = []
    regexToNfaHelper (Or re1 re2) n = ( NFA states alphabet delta begin final , next )
        where
        ( NFA s1 a1 d1 b1 f1, n1 ) = regexToNfaHelper re1 n
        ( NFA s2 a2 d2 b2 f2, n2 ) = regexToNfaHelper re2 n1
        states = s1 `union` s2 `union` [n2]
        alphabet = a1 `union` a2
        delta (st,sy)
            | st == n2 && isNothing sy = [b1] `union` [b2] -- epsilon-transitions from new
                start state to old start states
            | st == n2 = []
            | st `elem` s1 = d1 (st,sy)
            | st `elem` s2 = d2 (st,sy)
            | otherwise = []
        begin = n2
        final = f1 `union` f2
        next = n2+1
    regexToNfaHelper (Concat re1 re2) n = ( NFA states alphabet delta begin final , next )
        where
        ( NFA s1 a1 d1 b1 f1, n1 ) = regexToNfaHelper re1 n
        ( NFA s2 a2 d2 b2 f2, n2 ) = regexToNfaHelper re2 n1
        states = s1 `union` s2
        alphabet = a1 `union` a2
        delta (st,sy)
            | st `elem` f1 && isNothing sy = [b2] `union` d1 (st,sy) -- epsilon-transitions
                 from old NFA1's final states to NFA2's start state
            | st `elem` s1 = d1 (st,sy)
            | st `elem` s2 = d2 (st,sy)
            | otherwise = []
        begin = b1
        final = f2
        next = n2
    regexToNfaHelper (Star re1) n = ( NFA states alphabet delta begin final , next ) where
        (NFA s a d b f, n' ) = regexToNfaHelper re1 n
        states = s `union` [n']
        alphabet = a
        delta (st,sy)
            | st == n' && isNothing sy = [b] -- epsilon-transitions from new start to old
                start state
            | st == n' = []
            | st `elem` f && isNothing sy = [b] `union` d (st, Nothing) -- epsilon-
                transitions from final states also go back to old start state
            | otherwise = d (st,sy)
        begin = n'
        final = [n'] `union` f
        next = n'+1
```

## 3.2 Converting NFAs to regular expressions: Kleene's Algorithm

We implement Kleene's Algorithm to convert a given Nfa to an equivalent Regular-Expression.

```
module NfaToReg where

import DfaAndNfa ( NFA(NFA) )
import RegExp ( RegExp(..), orAll )


-- Get collection of labels in a NFA from a given origin state
-- to a destination state.
labelsFromTo :: (Eq state)
            =>  ((state, Maybe symbol) -> [state])                   -- Transition
                function
            -> [symbol]                                              -- Alphabet
            ->  state                                                -- Origin state
            ->  state                                                -- Destination
                state
            ->  [Maybe symbol]                                       -- Collection
                of labels
labelsFromTo delta labels o d = [label | label <- labels',
                                          d `elem` delta (o, label)]
                        where
                         -- labels' = lables \cup {\varepsilon}
                            labels' = fmap Just labels ++ [Nothing]
```

```haskell
-- Convert a label (or epsilon)
-- to a Reg-Ex
labelToReg :: Maybe symbol          -- label read
            -> RegExp symbol        -- Equivalent Reg-Ex
labelToReg Nothing = Epsilon
labelToReg (Just c)  = Literal c


-- COnverts a collection of labels
-- to a Reg-Ex in the obvious way, ie:
    -- labelsToReg ['a', 'b', 'c', \varepsilon]  = 'a' | 'b' | 'c' | '\varepsilon'
labelsToReg :: [Maybe symbol]       -- Collection of  labels \cup \varepsilon
            -> RegExp symbol        -- Equivalent Reg-Ex
labelsToReg labels = orAll (fmap labelToReg labels)

-- Reg-Ex of paths in NFA that go from
-- an origin state to a destination
-- without going through states >= a given state.
r :: (Eq state, Num state)
    =>  ((state, Maybe symbol) -> [state])                     -- Transition function
    -> [symbol]                                                -- Alphabet
    -> state                                                   -- Cannot go-through
        states >= this one
    -> state                                                   -- Origin state
    -> state                                                   -- Destination state
    -> RegExp symbol                                           -- Reg-Ex for all label-
        paths

-- R^{0} ij
r delta labels 0 i j
        --          =  a_{1} | ... | a_{m} | Epsilon
        | i == j    = labelsToReg (labelsFromTo delta labels i j) `Or` Epsilon

        --            =  a_{1} | ... | a_{m}         q_{j} in  \Delta{q_{i}, a_{1}) \cup
            ... \cup \Delta(q_{i}, a_{m})
        | otherwise
                        = labelsToReg (labelsFromTo delta labels i j)

--  R^{k} ij         = R^{k-1} ik               (R^{k-1} kk)*                   R^{k-1} kj
          |                R^{k-1} ij
r delta labels k i j = r' (k-1) i k  `Concat`   Star(r' (k-1) k k)  `Concat`    r' (k-1) k
    j    `Or`              r' (k-1) i j
                 where r' = r delta labels



-- Converts an NFA to an equivalent Reg-Exp
-- using kleene's algorithm.

-- NOTE: MAY NOT have right behvaiour for
-- state != Int
nfaToReg :: (Num state, Ord state)
         => NFA state symbol                                      -- NFA to convert
         -> RegExp symbol                                       -- Equivalent Reg-Ex
nfaToReg (NFA states labels delta start finals) = foldr (\f1 regExp -> r' n start f1  `Or`
    regExp) Empty finals
                where r' = r delta labels
                      n  = maximum states
```

## 4 Tests

```haskell
module Main where

import DfaAndNfa ( evaluateDFA, evaluateNFA, NFA )
import RegExp ( RegExp, matches, simplify )
import RegToNfa (regexToNfa)
import NfaToReg (nfaToReg)
import NfaToDfa (nfaToDfa)

import Test.Hspec ( hspec, describe, it )
import Test.QuickCheck ( Testable(property) )

main :: IO ()
main = hspec $ do
  describe "Regular languages: finite automata and regular expressions" $ do
    it "- simplify regex" $ property pSimplify
    it "- regex to nfa" $ property pRegexToNfa
    -- it "nfa to regex" $ property pNfaToRegex                    -- no Arbitrary NFA yet
    it "- regex to nfa and back" $ property pRegexToNfaAndBack     -- note that this might
        take very long
    -- it "- regex to nfa to dfa" $ property pRegexToNfaToDfa      -- sometimes fails

pSimplify :: RegExp Bool -> [Bool] -> Bool
pSimplify re s = matches s re == matches s (simplify re)

pRegexToNfa :: RegExp Bool -> [Bool] -> Bool
pRegexToNfa re s = matches s (simplify re) == evaluateNFA (regexToNfa $ simplify re) s

pNfaToRegex :: NFA Int Bool -> [Bool] -> Bool
pNfaToRegex nfa s = evaluateNFA nfa s == matches s (nfaToReg nfa)

pRegexToNfaAndBack :: RegExp Bool -> [Bool] -> Bool
pRegexToNfaAndBack re s = matches s (simplify re) == matches s ( (simplify . nfaToReg .
    regexToNfa ) re )

pRegexToNfaToDfa :: RegExp Bool -> [Bool] -> Bool
pRegexToNfaToDfa re s = matches s (simplify re) == evaluateDFA ( ( nfaToDfa . regexToNfa .
    simplify ) re) s
```

To run the tests, use `stack test`.

## 5 Conclusion

## References

[Sip12] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.