Finite automata and regular expressions in Haskell

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

Sunday 2nd June, 2024

Abstract

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

Contents

For this project, we have set three main goals: first, implement data types for deterministic and non-deterministic finite automata, as well as for regular expressions, while mainting as much generality as possible; second, implement conversions between them that preserve the language that they describe; third, test our construction using arbitrarily generated inputs. Regular languages are defined as those accepted by a deterministic finite automaton - if we look at the bigger picture, then, our goal is to implement the constructions that are used to prove that DFAs, NFAs and regular expressions have the same expressive power in describing regular languages. More specifically, we mostly follow Chapter 1 of [Sip12].

Our report is structured as follows. In Section 1, we provide an implementation of data types for DFAs and NFAs and we implement the powerset construction, which can be used to prove that DFAs have the same expressive power as NFAs. Next, in Section 2, we implement the data type for regular expressions. In Section 3, we implement the conversions from regular expressions to NFAs and from NFAs to regular expressions, which are used to show that regular expressions also describe regular languages. In Section 4, we describe our test suite. Finally, we provide a small demo in Section 5, and draw some conclusions in Section 6.

1 DFAs and NFAs

In this section we will define the data types DFA and NFA and discuss the ancillary functions that pertain to them.

1.1 Mathematical Definition and Haskell Implementation

The following are the mathematical definitions of DFAs and NFAs respectively.

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) Σ 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) Σ 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.

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

module DfaAndNfa where

import Data.Maybe ( fromMaybe, fromJust, isJust )
import Test.QuickCheck( Arbitrary(arbitrary), Gen, elements, frequency, listOf1, sublistOf,
     suchThat, vectorOf, chooseInt )
import Data.List (nub)

data DFA state symbol = DFA
                    { statesDFA :: [state]
                    , alphabetDFA :: [symbol]
                    , 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]
                    }
```

There are a couple of things to note about the implementation.

First, the types `state` and `symbol` are both arbitrary. That is, we can construct DFAs and NFAs with values of any type (though some functions might not perform as intended, given that they have some type constraint, we will mention these when they come up).

Second, notice how the δ^{DFA} function maps a tuple of type `state` and `symbol` to the type `Maybe state`. The reason for this is that δ^{DFA} can be a partial function, potentially leading to exceptions when executing functions call the transition function. To handle such exceptions more easily we implement δ^{DFA} to map to `Maybe state`, returning `Nothing` whenever the function is not defined for a particular combination of `(state, symbol)`. We make the necessary steps to and from the `Maybe` context within the functions requiring such conversions themselves.

Third, δ^{NFA} maps a tuple of type `state` and `Maybe symbol` to the type `[state]`, where the empty list is returned when the function is not defined for a `(state, Maybe symbol)` combination. We choose to represent $\Sigma \cup \{\varepsilon\}$ using `Maybe symbol` as it provides the additional value to the alphabet by which we can represent ε-transitions. Here too we make the conversion to and from `Maybe` within the functions that require these conversions themselves.

Below we detail the implementation of ancillary functions and instance declarations for the DFA and NFA data types.

1.2 Functions for DFAs and NFAs

Evaluate DFA and NFA

Below we define the function `evaluateDFA`, implemented from [Sip12].

```
evaluateDFA :: forall state symbol . Eq state => DFA state symbol -> [symbol] -> Bool
evaluateDFA dfa syms = case walkDFA (Just $ beginDFA dfa) syms of
    Nothing -> False -- If walkDFA returns Nothing at any point, evaluateDFA returns False
    Just s -> s ‘elem‘ finalDFA dfa   -- Otherwise, if walkDFA can make a transition for
        each symbol in the string,
                                      -- evaluateDFA checks whether the state it walked to
                                        is a sate in the list of final states.
    where -- 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 transitionDFA dfa (q,s) of
            Nothing -> Nothing -- If transitionDFA dfa (q,s) returns Nothing (i.e. it
                cannot make an s transition from state q), then the walkDFA returns Nothing
                .
            Just q’ -> walkDFA (Just q’) ss -- If transitionDFA dfa (q,s) returns Just q’,
                then we continue walkDFA with ss from q’.
```

The `evaluateDFA` function takes a specific DFA and checks whether the DFA accepts the list of symbols (of the same type as the symbols in the DFA’s alphabet) we give it. It does so by walking along the DFA according to the symbols in the list (by means of the `walkDFA` helper function) and checks whether the state it ends at is one of the final states of the DFA. Due to our use of the `‘elem‘` function, our `evaluateDFA` function has an equality constraint on the symbol type.

Next, we implement the same function for NFAs, the `eveluateNFA` function (implemented from [Sip12]). In this function we will have to consider the ε-closure for certain states, so we first define two functions called `epsilonClosure` and `epsilonClosureSet`. These functions will also figure in the conversion of NFAs to DFAs later on.

```
epsilonClosure :: forall state symbol . Eq state => NFA state symbol -> state -> [state]
epsilonClosure nfa x = closing [] [x] where
    closing :: [state] -> [state] -> [state]
    closing visited [] = visited -- visited acts as an accumulator which will be returned
        as the epsilon closed
                                 -- list of states once the function has gone through all
                                    the states it needs to close.
    closing visited (y:ys)
        | y ‘elem‘ visited = closing visited ys -- If y has already been visited we move on
        | otherwise = closing (y : visited) (ys ++ transitionNFA nfa (y, Nothing))
        -- otherwise we add y to the visited states and add all its
        -- epsilon related states to the yet to close list and recur the closing.

epsilonClosureSet :: Eq state => NFA state symbol -> [state] -> [state]
epsilonClosureSet nfa = concatMap (epsilonClosure nfa)

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 -- base case for the empty list of
        symbols, returns the epsilon-reachable states from the current set of states.
    walkNFA states (s:ss) = walkNFA (concatMap transition epsilonClosureStates) ss where
    -- recursively takes the epsilon-closure of the current set and finds all the s-
        reachable states from those
    -- and feeds it back into the walkNFA function.
        transition q = transitionNFA nfa (q, Just s) -- helper function for readability.
        epsilonClosureStates = epsilonClosureSet nfa states
```

The evaluateNFA function is quite similar to the evaluateDFA function. There are two notable differences, however.

First, because the transitionNFA function does not return Maybe state, but rather [state], we do not have to distinguish cases.

Second, the evaluateNFA function first takes the ε-closure of the current set of states before finding all the states that are reachable from each of those states by the corresponding symbol-transition. The epsilonClosure function recursively finds all the ε-reachable states from the initial state we want to close. It does so by finding the ε-reachable states for each element in the yet-to-close list of states and adding these to the list. It then adds the element to the accumulator list visited, marking the specific state as closed. It then recurs through the yet-to-close list (adding the results to the list and the closed element to visited each time, skipping over elements that have already been closed) until it has gone through the entire list, whereupon it returns the visited list. The function epsilonClosureSet extends this function to a list of states.

Because we make use of the `elem` function in both epsilonClosure and evaluateNFA, both of these functions also require an instance of Eq to be defined for the state type of the respective NFA.

Next, we define a pretty print function for both DFA and NFA.

Pretty Print for DFA and NFA

The following code implements pretty print functions for both DFAs and NFAs.

```
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 --    q -- s --> q'
    allTransitions = [((state, sym), fromJust $ transition (state, sym))
                     | state <- states,
                       sym <- alphabet,
                       isJust $ transition (state, sym) ]

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) ]
```

Each of these functions is essentially the same for both types. They both return a string that shows each part of the DFA/NFA, while nicely representing the transitions we can make from each state.

We will now move on to the `instance Show` and `instance Arbitrary` declarations for both the DFA and NFA data types.

1.3 Instance Declarations DFA and NFA

In this section, we detail the instance Show and instance Arbitrary declarations for DFA and NFA. Both of these instances need to be defined for both the DFA data type and the NFA data type for the quickCheck test-suite to function.

First, we discuss the implementation of `instance Show`.

Instance Show DFA and NFA

Below you find the code for `instance Show DFA`

```
instance (Show state, Show symbol) => Show (DFA state symbol) where
    show :: (Show state, Show symbol) => DFA state symbol -> String
    show dfa = "DFA {" ++
               "  statesDFA = " ++ show (statesDFA dfa) ++
               "  alphabetDFA = " ++ show (alphabetDFA dfa) ++
               "  transitionDFA = ‘lookup‘ " ++ show transitionListDFA ++
               "  beginDFA = " ++ show (beginDFA dfa) ++
               "  finalDFA = " ++ show (finalDFA dfa) ++
               "  }"
                where
                    -- Generates lookup table
                    transitionListDFA :: [((state,symbol), Maybe state)]
                    transitionListDFA = [ ((st, sy), transitionDFA dfa (st, sy))
                                        | st <- statesDFA dfa,
                                          sy <- alphabetDFA dfa ]
```

This implementation of instance Show DFA returns (as per convention) a string containing valid Haskell code, given our definition of the DFA data type. It generates a lookup table containing all the transitions we can make from a given state for a given symbol and prints the function by prepending `"‘lookup‘"` to this table, returning `Just s`, for some state s, whenever it finds an entry in the list for a specific `(state, symbol)` combination, and otherwise returning Nothing.

The implementation for `instance Show NFA` is almost identical to that for DFA. We again return a string containing valid Haskell code given our implementation of the NFA data type.

```
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) ++
               "  alphabetNFA = " ++ show (alphabetNFA nfa) ++
               "  transitionNFA = fromMaybe [] $ lookup " ++ show transitionListNFA ++
               "  beginNFA = " ++ show (beginNFA nfa) ++
               "  finalNFA = " ++ show (finalNFA nfa) ++
               "  }"
               where
                    -- Generates lookup table
                    transitionListNFA :: [((state, Maybe symbol), [state])]
                    transitionListNFA = [ ((st, sy), transitionNFA nfa (st, sy))
                                        | st <- statesNFA nfa,
                                          sy <- Nothing : map Just (alphabetNFA nfa) ]
```

Here too we generate a lookup table containing all the possible transitions we can make from any given state. We ensure that `transitionNFA` is a valid Haskell function by prepending "`fromMaybe [] $ lookup` " to the table. The function will then return `s` if the `lookup` function returns `Just s`, for some state s, and returning the empty list (`[]`) if the lookup function returns Nothing.

Instance Arbitrary DFA and NFA

We now move on to our implementation of instance Arbitrary DFA and NFA. These instances are essential to the quickTest test-suite, as they dictate how the arbitrary DFAs and NFAs are generated during the testing procedure.

We begin by taking a closer look at our implementation of `instance Arbitrary DFA`.

```
instance (Arbitrary state, Arbitrary symbol, Eq state, Eq symbol, Num state, Ord state) =>
    Arbitrary (DFA state symbol) where
    arbitrary :: (Arbitrary state, Arbitrary symbol, Eq state, Eq symbol) => Gen (DFA state
         symbol)
    arbitrary = do
            states <- nub <$> listOf1 (arbitrary :: Gen state) -- generates a nonempty list
                 of arbitrary states
            alphabet <- uniqueAlphabet -- generates a list of length 2 of unique arbitrary
                symbols
            transition <- randomTransitionDFA states alphabet -- generates the arbitrary
                transition function with the appropriate type
            begin <- elements states -- takes an random element in the list of states to be
                 the begin state
            final <- sublistOf states `suchThat` (not . null) -- takes a nonempty sublist
                of the states to be designated final states
            return $ DFA states alphabet transition begin final
        where
            -- helper function to generate the list of unique arbitrary sybols, always has
                length 2
            uniqueAlphabet = do
                x <- (arbitrary :: Gen symbol)
                y <- (arbitrary :: Gen symbol) `suchThat` (/= x)
                return [x, y]
            -- helper function to generate the transition function of arbitrary DFA
            randomTransitionDFA states alphabet = do
                st <- (nub <$> listOf1 (elements states)) `suchThat` (not . null) --
                    generates a non-empty list consisting of elements of the list of states
                syms <- vectorOf (length st) (elements alphabet) -- generates a vector of
                    the length of st consisting of the (possibly duplicate) elements of the
                     alphabet
                st' <- listOf1 (elements states) -- generates a non-empty list consisting
                    of (possibly duplicate) elements of the list of states
                let transitionTable = zip (zip st syms) st' -- creates the transistion
                    table
                return $ \(state, symbol) -> lookup (state, symbol) transitionTable
```

For each of the arguments of the DFA constructor we define how to arbitraryly generate the right value. For the states we generate a non-empty list of the state type of the DFA which scales with the complexity of the test. For the alphabet we generate a list of length 2 of unique values. The begin state is a random element chosen from the list of states and the final states are a randomly chosen subset. The only intricate part of the function is the arbitrary transition function generation. To this end we first generate a lookup table. This is done by generating a non-empty list consisting of unique elements in the list of states, then generating a list of (possibly duplicate) symbols in the alphabet of the same length and zipping these two to create a list of tuples of the familiar form `(state, symbol)`. These tuples will figure as the arguments for our transition function. We generate the values these tuples will map to by generating yet another non-empty list of (possibly duplicate) elements in the list of states and zipping the list of tuples with this list to generate the final lookup table. The transition function is then generated by returning `(state, symbol) -> lookup (state, symbol) transitionTable`.

Because we make use of the `nub` function twice, once when returning the list of states and once when generating a unique list of states for the transition function, our implementation of Arbitrary for DFA has an `Eq` constraint on the type of the states of the DFA. We also have this constraint for the type of the symbols of the DFA, this time because we have to compare the symbols in the alphabet to make sure it consists of two unique symbols.

The implementation for `instance Arbitrary NFA` is as follows.

```
instance (Arbitrary symbol, Eq symbol) => Arbitrary (NFA Int symbol) where
    arbitrary :: (Arbitrary symbol, Eq symbol) => Gen (NFA Int symbol)
    arbitrary = do
            n <- chooseInt (2,5) -- generates a random element n in the range (2,5).
            let states = [1..n] -- sets the states to the list of 1 to n.
            alphabet <- uniqueAlphabet -- generates a list of length 2 of unique arbitrary
                symbols
            transition <- randomTransitionNFA states alphabet -- generates the arbitrary
                transition function with the appropriate type
            begin <- elements states -- takes an random element in the list of states to be
                 the begin state
            final <- sublistOf states `suchThat` (not . null) -- takes a nonempty sublist
                of the states to be designated final states
            return $ NFA states alphabet transition begin final
        where
            uniqueAlphabet = do
                x <- (arbitrary :: Gen symbol)
                y <- (arbitrary :: Gen symbol) `suchThat` (/= x)
                return [x, y]
            -- helper function to generate the transition function of arbitrary NFA
            randomTransitionNFA states alphabet = do
                st <- (nub <$> listOf1 (elements states)) `suchThat` (not . null)  --
                    generates a non-empty list consisting of unique elements of the list of
                     states
                syms <- vectorOf (length st) $ frequency [(1, return Nothing), (9, elements
                     (map Just alphabet))] -- generates a vector of the length of st where
                    the elements are either Nothing or a Just element in the alphabet
                stList <- listOf1 $ sublistOf states -- generates a non-empty list
                    consisting of subsets of the list of states
                let transitionTable = zip (zip st syms) stList -- creates the transistion
                    table
                return $  \(state, symbol) -> fromMaybe [] $ lookup (state, symbol)
                    transitionTable
```

As you can see, the implementation for `instance Arbitrary NFA` is quite similar to that for DFA. There are two main differences.

First, rather than generating a list of states of an arbitrary type that scales with the complexity of the tests, here we generate a list of states that is constrained to the `Int` type and is limited to a maximum length of 5. The first constraint is due to the fact that our `nfaToReg` function (which we will detail later), only works on states of the integer type. The second constraint is to limit the complexity of the test cases, where too large of a list of states might make it so that our test-suite takes too long to complete all of its tests.

Second, during the generation of the lookup table for the transition function the list of symbols to construct the `(state, symbol)` tuple is generated from a 1 to 10 distribution of occurrences of Nothing (representing the ε-transitions) and 9 to 10 occurrences of elements in the alphabet. In this process also, rather than generating a list of single elements from the list of states, we generate a list of subsets of the list of states as target values for the `(state, symbol)` to be mapped to. This way the function `(state, symbol) -> fromMaybe [] $ lookup (state, symbol) transitionTable` returns a subset of the list of states.

For the same reason as with the implementation of `instance Arbitrary DFA`, our implementation of `instance Arbitrary NFA` also has the `Eq` type constraint for both the NFA `state` type and `symbol` type.

This concludes our implementation of the DFA and NFA data types and their ancillary functions.

1.4 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 ( mapMaybe, fromJust, isJust )
import Data.List ( intersect, nub, sort )
```

We straight forwardly implement the powersetconstruction . Here, we translate a NFA, $N = (Q, \Sigma, \delta, q_0, F)$, where Q is the set of states, Σ is the alphabet, δ is the transition function $\delta : Q \times \Sigma \to \mathcal{P}(Q)$, $q_0 \in Q$ the initial state and $F \subseteq Q$ the set of final states. We define the corrsponding DFA as $D = (Q', \Sigma, \delta', q, F')$ where

- $Q' = \mathcal{P}(Q)$
- $\delta' : Q' \times \Sigma \to Q',\ \delta'(S, x) = \delta'(\bigcup_{q \in S}\{\delta(q, x)\}, \varepsilon)$
- $q = \delta(q_0, \varepsilon)$
- $F' = Q' \cap F$

Instead of using sets for the Powerset construction, we will use lists. Therefore, we have to take care of sorting the list to not run into problems evolving from `[0,1]` not being the same as `[1,0]`. While the definition of the alphabet, initial and acceptance states is straightforward, the definition of the transition function is a bit more involved. The problems when implementing this mainly arise from having partial functions as transitions. After presenting the `nfaToDfa` function, we will further elaborate on this.

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

nfaToDfa :: (Eq state, Ord state) => NFA state symbol -> DFA [state] symbol
nfaToDfa (NFA statesN alphabetN transN startN endN) =
  let nfa = NFA statesN alphabetN transN startN endN
      statesD = map sort $ powerSetList statesN
                                                           -- new set of states
      alphabetD = alphabetN
                                                                                -- same
          alphabet as the NFA
      startD = sort $ 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 $ sort $ 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
```

The function `transD` takes a state `sf` in the new DFA (which is a list of states in the original NFA) and a symbol `sy` and returns a state in the DFA (also a list). First, the lambda function `s -> transitionNFA nfa (s, Just sy)` is concatmapped over `st`. This gives us a list of all states reachable from the states in `st` by a `sy`-transition. In the next step, we have to add all states reachable by a ε-transition as these are, in the original NFA, reachable without reading a symbol. In the original algorithm we would be done here, but as we use lists instead of sets, we have to apply the functions `nub` and `sort` to make mirror the behaviour of sets in the sense that two sorted and nubbed lists are equal when they have the same elements in them.

The proof that the resulting DFA accepts exactly the same strings as the original NFA works by induction on the length of the input string and is almost completely represented in the definitions of the translation. The base case follows because the initial state in the DFA is the epsilon closure of the original initial staes which are exactly the states reachable given the empty string as input. This mirrors the definition of the initial state and endstate. The induction step uses that the states one can reach after reading a symbol x is the ε-closure of the set of x-reachable states. This is mirrored by the two steps in the definition of `transD`. The complete proof can be found in any text book on automata theory. See for instance Theorem 1.39 from [Sip12].

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 initial states. Then, we restrict the transitions and final states to the reachable states in the original DFA.

```haskell
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

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), fromJust $ transitionDFA dfa (s, a)) | s <-
      reachableStates, a <- alphabetDFA dfa, isJust $ transitionDFA dfa (s, a) ]
  newTransition (s, a) =  lookup (s,a) transitionsToReachables
  newFinalStates = filter (`elem` reachableStates) (finalDFA dfa)
```

2 Regular Expressions

In this section, we implement the data type for regular expressions, as well as some useful functions for matching and simplyfing, and define an `Arbitrary` instance for our data type to generate random regular expressions for our test suite.

Definition 3. Fix an alphabet Σ. We say that R is *regular expression* over Σ 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 `printRE` for displaying regular expressions in a more readable format[1].

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

[1]This technically operates under the assumption that the alphabet does not contain `*` or `+` or the parentheses symbolbols, which would make the `printRE` 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.

```haskell
module RegExp where

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

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

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

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

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

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

Formally, the language described by a regular expression R over Σ is denoted $L(R)$ and consists exactly of the strings over Σ 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 Σ. 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 $s = a$;

(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.

```haskell
matches :: Eq symbol => [symbol] -> RegExp symbol -> 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 simple algebraic identities, that are stated as comments in the code for compactness. Note that this function does not minimize a given regular expression[2] but it is useful, as a simple heuristic, 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. The `simplify` function works by repeatedly applying a simple one-step simplification function until no further simplifications are possible.

```
simplify :: Eq symbol => RegExp symbol -> RegExp symbol
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 symbol => RegExp symbol -> RegExp symbol
        oneStepSimplify Empty = Empty
        oneStepSimplify Epsilon = Epsilon
        oneStepSimplify (Literal l) = Literal l
        oneStepSimplify (Or re1 re2)
            | re1 == Empty = oneStepSimplify re2    -- Empty | re2 -> re2
            | re2 == Empty = oneStepSimplify re1
            | re1 == re2 = oneStepSimplify re1      -- re1 | re1 -> re1
            | otherwise = Or (oneStepSimplify re1) (oneStepSimplify re2)
        oneStepSimplify (Concat re1 re2)
            | re1 == Empty || re2 == Empty = Empty  -- Empty `Concat` re -> Empty
            | re1 == Epsilon = oneStepSimplify re2  -- Epsilon `Concat` re2 -> re2
            | re2 == Epsilon = oneStepSimplify re1
            | otherwise = Concat (oneStepSimplify re1) (oneStepSimplify re2)
        oneStepSimplify (Star re') = case re' of
            Empty -> Epsilon                                -- Empty* -> Epsilon
            Epsilon -> Epsilon                              -- Epsilon* -> Epsilon
            Or Epsilon re2 -> Star (oneStepSimplify re2)    -- (Epsilon | re2)* -> (re2)*
            Or re1 Epsilon -> Star (oneStepSimplify re1)
            Star re1 -> Star (oneStepSimplify re1)          -- ((re1)*)* -> (re1)*
            _ -> Star (oneStepSimplify re')
```

Finally, we implement a way to generate random regular expressions using QuickCheck. We try to keep their size relatively small so that testing that converting back and forth from regular expressions to NFAs does not take too long.

```
instance Arbitrary symbol => Arbitrary (RegExp symbol) where
  arbitrary :: Arbitrary symbol => Gen (RegExp symbol)
  arbitrary = sized randomRegExp where
    randomRegExp :: Int -> Gen (RegExp symbol)
    randomRegExp 0 = oneof [ Literal <$> (arbitrary :: Gen symbol), return Epsilon, return
        Empty ]
    randomRegExp n = oneof [ Literal <$> (arbitrary :: Gen symbol), 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)
                        ]
```

[2]This is a very hard computational problem, and implementing a solution for it is outside the scope of our project.

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.*

Using the fact that DFAs have the same expressive power as NFAs, we will implement conversions from regular expressions to NFAs and back. In particular, in § 3.1, we implement the construction of an NFA from a regular expression which can be used to formally prove 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 via Kleene's algorithm, which shows 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 construction of 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 Σ and let R be a regular expression over Σ. The proof is by induction on the structure of R. The basic idea is to construct the simplest possible NFAs for the base cases of R, and then make clever transformations to the NFAs given by the inductive hypothesis for the inductive cases. We give the full details only of some cases for brevity.

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.

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 $L(R_2)$.

Case $R = R_1^*$. We build a new NFA N by adding 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. This is to guarantee that N accepts ε. Moreover, we add epsilon-transitions from N_1's final states to N_1's start state. This is to simulate the fact that $*$ stands for "arbitrary number of repetitions of the pattern". □

The implementation of the construction described in the proof is very straightforward, with only a couple technical 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 symbol => RegExp symbol -> NFA Int symbol
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 symbol => RegExp symbol -> Int -> (NFA Int symbol, 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

Here we implement the construction of the proof of the following.

Lemma 7. *If a language is regular, then it is described by a regular expression.*

```
module NfaToReg(nfaToReg) where

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

We implement Kleene's Algorithm to convert a given NFA to an equivalent regular expression.

First, given a transition function `delta`, an alphabet `labels`, a start state `o` and an end state `d`, we compute `labelsFromTo delta labels o d`, i.e. the collection of labels/arrows that take us from `o` to `d` in our NFA.

```
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 {\epsilon}
                            labels' = fmap Just labels ++ [Nothing]
```

Then, for a given label (or ε-label) we trivially compute the regex for it using `labelToReg`.

```

labelToReg :: Maybe symbol           -- label read
            -> RegExp symbol         -- Equivalent regex
labelToReg Nothing = Epsilon
labelToReg (Just c)  = Literal c
```

We then trivially extend this to a list of labels using `labelsToReg`, for example:

$$\texttt{labelsToReg [a, b, c, } \varepsilon\texttt{]} \;=\; \texttt{a | b | c | } \varepsilon.$$

```
labelsToReg :: [Maybe symbol]        -- Collection of labels
            -> RegExp symbol         -- Equivalent regex
labelsToReg labels = orAll (fmap labelToReg labels)
```

Finally, we are now ready to define our helper function `r` which is the key to our translation. Note, however, that `r` forces two restrictions on our NFA:

1. Need `state == Int` to preform induction on a state
2. For our list of states, need `states == [1,2,` $\cdots$ `, n]`

To ensure these restrictions, we define `correctStates states` to check `states == [1,2,`$\cdots$`,n]`.

```
correctStates :: [Int] -> Bool
correctStates states = states == [1..n] where n = length states
```

Now, for our helper function `r`: for `i, j` $\in$ `[1,`$\cdots$`,n]` and `k` $\in$ `[0,1,`$\cdots$`,n]`:
"`r k i j`" means "All paths in NFA from `i` to `j` where all intermediate-states are $\leq$ `k`"

For example, "`r 2 1 3`" would accept the path

$$1 \to 2 \to 1 \to 3$$

and reject the path

$$1 \to 2 \to 1 \to 3 \to 3$$

We define this by induction on upperbound `k` as follows.

- "Direct labels from `i` to `i` OR do nothing":
  $\texttt{r}^{0}$ `i i = labelsToReg(labelsFromTo delta labels i i) |` ε
- "Direct labels from `i` to `j`":
  $\texttt{r}^{0}$ `i j = labelsToReg(labelsFromTo delta labels i j)`
- "Take a `k-1`-bounded path that passes through `k` OR take one that does not pass through `k`":
  $\texttt{r}^{\texttt{k}}$ `i j =` $\texttt{r}^{\texttt{k-1}}$ `i k ·` ($\texttt{r}^{\texttt{k-1}}$ `k k)* ·` $\texttt{r}^{\texttt{k-1}}$ `k j |` $\texttt{r}^{\texttt{k-1}}$ `i j`

So in conclusion our code for `r` is the following.

```
r :: ((Int, Maybe symbol) -> [Int])        -- Transition function
    -> [symbol]                             -- Alphabet
    -> Int                                  -- All intermediate-states <= this bound
    -> Int                                  -- Origin state
    -> Int                                  -- Destination state
    -> RegExp symbol                        -- Reg-Ex for all label-paths

r delta labels 0 i j
        | i == j    = labelsToReg (labelsFromTo delta labels i j) `Or` Epsilon
```

```
         | 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
```

Finally, to compute our equivalent regex for `NFA [1,···,n] labels delta start finals`, we define it as:

$$\bigcup_{\texttt{f1} \in \texttt{finals}} \texttt{r n start f1}$$

Thus, we implement `nfaToReg` as follows.

```
nfaToReg :: NFA Int symbol                                    -- NFA to convert
         -> RegExp symbol                                     -- Equivalent Reg-Ex
nfaToReg (NFA states labels delta start finals) =
        -- Need states == [1,2,..n] for
        -- helper function r!
        case correctStates states of
            False -> error "states is not == [1, 2,..., n]"
            -- Compute Or_{f1 in finals} r n start f1
            True  -> foldr (\f1 regExp -> r' n start f1  `Or` regExp) Empty finals
                where r' = r delta labels
                      n  = maximum states
```

4 Tests

For our test suite, we use our `Arbitrary` implementations of DFAs, NFAs and regular expressions to test whether our conversions preserve the language that the automaton or the regular expression describes. We also test whether they compose - for instance, whether applying `regToNfa` and then `nfaToReg` to an arbitrary regular expression results in an equivalent regular expression. We arbitrarily choose `Int` as our state type and `Bool` as our symbol type for simplicity, and because they also both satisfy all constraints that our functions impose on the `state` and `symbol` type.

Note that in our test suite, we take for granted that our `matches` function works, without testing it explicitly. We think this is reasonable, because the function consists of one of the most basic straightforward direct implementations of the mathematical definition. We however also tested it manually on small manually generated inputs, which we do not feature here. Moreover, we make use of the `simplify` function for regular expressions to speed up the tests.

Finally, it is worth noting that we often had to limit the size of our arbitrarily generated strings so that our test suite would not take too long to execute. These limits were simply set through several test runs.

```
main :: IO ()
main = hspec $ do
  describe "Regular languages: finite automata and regular expressions" $ do
    prop "- simplify regex" $ \(re :: RegExp Bool) s -> length s <= 100
                                ==> matches s re == matches s (simplify re)
    prop "- regex to nfa" $ \(re :: RegExp Bool) s -> length s <= 100
                                ==> matches s (simplify re) == evaluateNFA (regexToNfa $
                                    simplify re) s
    prop "- nfa to regex" $ \(nfa :: NFA Int Bool) s -> length s <= 10
```

```
                                ==> evaluateNFA nfa s == matches s (simplify $ nfaToReg nfa
                                    )
    prop "- regex to nfa and back" $ \(re :: RegExp Bool) s -> length s <= 20
                                ==> matches s (simplify re) == matches s ( (simplify .
                                    nfaToReg . regexToNfa ) re )
    prop "- nfa to regex and back" $ \(nfa :: NFA Int Bool) s -> length s <= 10
                                ==> evaluateNFA nfa s == evaluateNFA ((regexToNfa .
                                    simplify . nfaToReg) nfa) s
    prop "- regex to nfa to dfa" $ \(nfa :: NFA Int Bool) s -> length s <= 10
                                ==> evaluateNFA nfa s == evaluateNFA ((regexToNfa .
                                    simplify . nfaToReg) nfa) s
    prop "- nfa to dfa" $ \(nfa :: NFA Int Bool) s -> length s <= 25
                                ==> evaluateNFA nfa s == evaluateDFA (nfaToDfa nfa) s
    prop "- minimize dfa" $ \(dfa :: DFA Int Bool) s -> length s <= 50
                                ==> evaluateDFA dfa s == evaluateDFA (
                                    removeUnreachableStates dfa) s
```

5 A playground for finite automata and regular expressions

Here, we manually define a simple DFA and a simple NFA to show some of the most important parts of our code.

```
testDFA :: DFA Int Char -- this accepts the language (a*)b
testDFA = DFA [1,2]
              "ab"
              (`lookup` [((1,'a'), 1), ((1,'b'), 2)])
              1
              [2]

testNFA :: NFA Int Char -- this accepts all the strings over {a,b}
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]
```

It is easy to see that `testDFA` accepts the language a^*b, while `testNFA` accepts all the strings over the alphabet $\{a, b\}$.

```
main :: IO ()
main = do
  putStrLn "\nWelcome to our demo! \n"
  putStrLn "--- test DFA ---"
  putStrLn $ printDFA testDFA
  putStrLn "--- testNFA ---"
  putStrLn $ printNFA testNFA
  putStrLn "--- testNFA to DFA ---"
  putStrLn $ printDFA $ (removeUnreachableStates . nfaToDfa) testNFA
  putStrLn "--- testNFA to regex ---"
  putStrLn $ printRE $ (simplify . nfaToReg) testNFA
  putStrLn "--- testNFA to regex and back ---"                                     -- note
       that this is quite large already!
  putStrLn $ printNFA $ regexToNfa $ (simplify . nfaToReg) testNFA
  -- putStrLn "--- testNFA to regex and back and to DFA ---"                       -- this
       might take VERY long
  -- putStrLn $ printDFA $ nfaToDfa $ regexToNfa $ (simplify . nfaToReg) testNFA
```

We can run this program with the commands `stack build && stack exec myprogram`.

6 Conclusion

In this project, we have implemented data types for regular expressions, deterministic automata, and non-deterministic automata. Using these, we implemented the constructions that are used to prove important results in automata theory: the expressive equivalence between regular languages, NFAs, and DFAs.

Haskell's type system and the `Maybe` monad allowed us to effectively model partial transition functions for DFAs by using `Maybe state` to map arguments that have no value specified, meaning no transition for some symbols, to a dummy "garbage" state. The only downside is that the models are a bit more complicated to write, and the code for `evaluateDFA` is slightly more complicated, which we think is an acceptable trade-off. Here, one could add a a translation function that transforms a given input list to a function of the right type. As for NFAs, using the `Maybe` type for symbols allowed us to separately treat the ε-transitions for NFAs by singling out an object in a non-specified type to work as ε.

Currently, the NFA to DFA translation is split into two parts: first, we translate the DFA using the powerset construction, and then minimize it. By producing the minimized DFA on-the-fly and generating only the states we can transition to without computing the entire power set, we could improve the efficiency of the translation.

As for translating from regular expressions to NFAs, this was straightforward: "glue" together the recursively generated NFAs while keeping track of fresh state labels to use for new states, using `Int` as our state type. In contrast, the other direction was much trickier, requiring the constraint of `Int`-labeled states for the helper function `r k i j` defining all `k`-bounded paths from `i` to `j`. Thus, our translations between regular expressions and NFAs are both limited to the case where states are labled with `Int`. We believe this is again an acceptable trade-off to have simpler code, as the type used for states only acts as a labelling scheme for states in our functions. If however in the future we wished to extend this to NFAs with other state labels, we could write simple coercions between arbitrary and `Int` labels and work "under the hood" solely with the `Int` labels.

Our test suite shows that the algorithms work in many cases, but due to the inefficiency of some of the algorithms, such as the translation `nfaToReg`, we cannot test for long expressions.

References

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