Haskelleene: A Very Haskell Implementation of Regular Expression and Finite Automaton Equivalence

Liam Chung, Brendan Dufty, Lingyuan Ye

Sunday 2nd June, 2024

Abstract

In this project, we implement finite (non)deterministic automata and regular expressions, with their corresponding semantics of regular languages. We also implement the conversions between these different structures, including power-set determinisation and Kleene's algorithm. Lastly, we use QuickCheck to verify the behavioural equivalence of automata and regular expressions under these constructions, as stated by Kleene's Theorem.

Contents

1 Implementing Automata and Regular Expressions

1.1 The Alphabet

In this section we define our most basic data structure: a finite input alphabet. Our current implementation choice is to record alphabet as a *type class*, equipped with a complete list of symbols:

```
module Alphabet where

import Data.List ( sort )
import Test.QuickCheck ( elements, Arbitrary(arbitrary) )

class Ord a => Alphabet a where
  completeList :: [a]
  pPrintLetter :: a -> String
```

The reason for this implementation choice is that we can silently pass this recorded list of complete alphabet as input via constraint declarations. We also require any alphabet shall be ordered.

Here is an example: The function `alphIter` will check if a list contains exactly each element of the alphabet once. This function will be useful when we work with deterministic automata.

```
alphIter :: Alphabet a => [a] -> Bool
alphIter l = sort l == completeList
```

The main input alphabet we are going to use on testing consists of three letters. This choice is of course not essential to our main implementation, which will be parametric over all type instances of the class `Alphabet`.

```
data Letter = A | B | C deriving (Eq, Ord, Show)

instance Alphabet Letter where
  completeList = [A,B,C]
  pPrintLetter A = "a"
  pPrintLetter B = "b"
  pPrintLetter C = "c"
```

To use the QuickCheck library to test on arbitrary inputs, we also define an instance of `Arbitrary` for our type `Letter`:

```
instance Arbitrary Letter where
  arbitrary = elements completeList
```

1.2 Deterministic and Non-Deterministic Automata

Our first major goal is to implement (non)deterministic automata. Recall that an automaton with input alphabet Σ consists of a set of states, a subset of accepting states, and a transition function δ. As input, δ takes a symbol of the alphabet, and a current state of the automaton. For a deterministic automaton, it outputs a unique next state; For a non-deterministic automaton, it outputs a *list* of possible next states.

```
{-# LANGUAGE TupleSections #-}

module Automata where

import Alphabet ( Alphabet(..), alphIter ) -- contains all of our utility functions
import Data.Maybe ( isJust, fromJust, isNothing )
import Data.List ( nub, intersect )
import qualified Data.Set as Set
```

However, since δ of an automaton is a *function*, it is not directly definable as inputs. Our implementation choice is to record the *transition table* of the transition function, and use the following types as an interface of encoding and decoding an automata with finite inputs:

```
type TDict l s = [(s, [(Maybe l, s)])]

data AutData l s = AD { stateData :: [s]
                      , acceptData :: [s]
                      , transitionData :: TDict l s }

pPrintAD :: (Alphabet l, Show s, Eq l, Eq s) => AutData l s -> String
pPrintAD ad = unlines $ ["States:" ++ showSts ad (stateData ad),"","Transitions:"]
                        ++ (concatMap transitions . transitionData) ad where
    showSts d = foldr (\s l -> " " ++ show s ++ isAccept s d ++ l) ""
    isAccept s d = if s `elem` acceptData d then "*" else ""
    transitions (s,trs) = map (stTrs s) trs
    stTrs s t = show s ++ " --" ++ letter ++ "> " ++ output where
      letter | isNothing (fst t) = "Em"
             | otherwise = (pPrintLetter . fromJust . fst) t ++ "-"
      output = show $ snd t
```

Here `l` should be thought of as the type of the chosen alphabet, while `s` is the type of our states. The type `TDict` then acts as the type of transition tables. A pair in `TDict` should be thought of recoding the information of given a current state, what are the possible output states of a given input. The fact that we have used `Maybe l` is that we want the type `AutData` to be simultaneously able to record both deterministic and non-deterministic automata, thus with the possibility of ϵ-transitions. For some examples of using `AutData` to encode automata, see Section 3.1.

As mentioned, the data type of deterministic automata should be defined as follows:

```
data DetAut  l s = DA { states :: [s]
                      , accept :: [s]
                      , delta :: l -> s -> s }
```

A consequence of using the same data type (`AutData`) to encode both deterministic and non-deterministic automata is that we need to check whether the data defines a deterministic automaton. The following are some useful utility functions to check whether the given transition table has certain properties:

```
-- intended to be used as aut `trsOf` s, to see what transitions s has in aut
trsOf :: Eq s => AutData l s -> s -> [(Maybe l, s)]
trsOf aut s
  | isNothing $ lookup s $ transitionData aut = []
  | otherwise = fromJust $ lookup s $ transitionData aut

-- "all unique"
allUnq:: Eq a => [a] -> Bool
allUnq = unqHelp [] where
   unqHelp _ [] = True
   unqHelp seen (x:xs) = notElem x seen && unqHelp (x:seen) xs

appendTuple :: ([a],[b]) -> ([a],[b]) -> ([a],[b])
appendTuple (l,l') (m,m') = (l++m,l'++m')
```

Now we can encode `AutData` as a deterministic automaton, so long as the data it describes *can* do so:

```
-- contingent on passing detCheck, turn data into DA
encodeDA :: (Alphabet l, Eq s) => AutData l s -> Maybe (DetAut l s)
encodeDA d | not $ detCheck d = Nothing
           | otherwise = Just $ DA { states = stateData d
                                   , accept = acceptData d
                                   , delta = safeDelta } where
    tData = transitionData d
    safeDelta ltr st = (fromJust . lookup (Just ltr) . fromJust . lookup st) tData      --
        convert data to delta functionp
    detCheck ad = length (fst <$> tData) == length (stateData ad) && allUnq (fst <$> tData) --
        all states are in transitionData exactly once
              && all (detCheckTr . snd) tData
    detCheckTr trs = notElem Nothing (fst <$> trs)        -- no empty transitions
                     && alphIter (fromJust . fst <$> trs) -- transition set is exactly the
                         alphabet
```

We end the basic implementation of deterministic automata by providing its semantic layer, i.e. on a given input and an initial state, whether a deterministic automata accept a list of symbols or not. Intuitively, the `run` function uses the transition function to trasverses an input string on an automaton and output the terminating state. Then `acceptDA` tests whether the input is accepted by testing whether the terminating state is an accepted state:

```
run :: DetAut l s -> s -> [l] -> s
run _ s0 [] = s0
run da s0 (w:ws) = run da (delta da w s0) ws

acceptDA :: (Eq s) => DetAut l s -> s -> [l] -> Bool
acceptDA da s0 w = run da s0 w `elem` accept da
```

We now proceed to implement non-deterministic automata, in a very similar way. The only difference is that the transition function now also accepts empty input, viz. the so-called ϵ-*transitions*, and the result of a transition function is a list of all possible next states.

```
data NDetAut l s = NA { nstates :: [s]
                      , naccept :: [s]
```

```
                      , ndelta :: Maybe l -> s -> [s] }

-- all DAs are also NAs: the trivial forgetful functor DA -> NA
fromDA :: (Alphabet l) => DetAut l s -> NDetAut l s
fromDA da = NA { nstates = states da
               , naccept = accept da
               , ndelta = newDelta } where
  newDelta Nothing _ = []
  newDelta (Just l) st = [delta da l st]
```

It will be convenient for us to be able to convert `NDetAut`s back into `AutData`, as in some contexts working with the explicit data is easier, for example Kleene's algorithm (covered in Section 2).

```
decode :: (Alphabet l, Eq s) => NDetAut l s -> AutData l s
decode nda = AD { stateData = sts
                , acceptData = naccept nda
                , transitionData = trandata }
  where sts = nstates nda
        ntrans = ndelta nda
        symlist = Nothing : (Just <$> completeList)
        trandata = graph help sts
        help st = concatMap (\sym -> (sym,) <$> ntrans sym st) symlist
        graph f as = zip as $ f <$> as
```

As an aside, we can now actually pretty print both `DetAut` and `NDetAut`, by decoding them to `AutData` and using the pretty print function we defined for that:

```
pPrintNA :: (Alphabet l, Show s, Eq l, Eq s) => NDetAut l s -> String
pPrintNA = pPrintAD . decode

pPrintDA :: (Alphabet l, Show s, Eq l, Eq s) => DetAut l s -> String
pPrintDA = pPrintAD . decode . fromDA
```

Now we can encode NAs from `AutData`, and there's no need to fear the data being invalid; though we will want to reformat the data if it's not organised properly, for example listing the potential transitions from a state split into different parts of the `TDict`. We also ignore ϵ loops, as they do not alter the behaviour of the automaton and create unnecessary complexities for the semantic algorithm.

```
-- make data into an NA (if data formatted properly; otherwise, format then encode)
encodeNA :: (Alphabet l, Eq s) => AutData l s -> NDetAut l s
encodeNA d = NA { nstates = stateData d
                , naccept = acceptData d
                , ndelta = newDelta } where
  newDelta sym st = case lookup st tData of
                      Nothing -> []
                      Just ls -> nub [ st' | (sym', st') <- ls, sym' == sym, isJust sym' || st'
                           /= st ]
  tData = if trsMerged rawTData then rawTData else mergeTrs rawTData
  trsMerged = allUnq . map fst
  rawTData = transitionData d
  -- slow, so we don't always want to be calling this
  mergeTrs [] = []
  mergeTrs ((tr0,tr1):trs) = mTr:mergeTrs remTrs where
    mTr = (tr0, fst prop ++ tr1)
    remTrs = snd prop
    prop = propTrs tr0 trs
  -- for a given state, propagate all of its outputs together
  propTrs _ [] = ([],[])
  propTrs st (tr:trs) = appendTuple resultTuple (propTrs st trs) where
    resultTuple = if st == fst tr then (snd tr,[]) else ([],[tr])
```

Now we implement the semantic layer for non-deterministic automata. The algorithm used for implementing `runNA` for traversing an input string on a non-deterministic automaton is inspired by [Cox07]. We keep a set of *active states*, along with what is left of the word in their trace. We use `oneStep` to simulate traversing an arrow in the automaton. We run, one step at a time, updating our set of active states, and stopping when we reached a fixed point.

```
runNA :: (Alphabet l, Ord s) => NDetAut l s  -> s -> [l] -> [([l], s)]
runNA na st w = (Set.elems . runNAFixPt na . Set.map (w,)) start
  where start = if null w then Set.singleton st else Set.fromList $ map snd $ runNA na st []
        runNAFixPt nda active = if active == next then active else runNAFixPt na next
          where next = appStep active
                appStep ac = Set.fromList $ concatMap (uncurry oneStep) ac
                oneStep [] s = ([], st) : (([],) <$> (ndelta nda Nothing s))
                oneStep (c:cs) s  = ((cs,) <$> (ndelta nda (Just c) s))
                                    ++ ((c:cs,) <$> (epReachable nda s))

epReachable :: (Alphabet l, Ord s) => NDetAut l s -> s -> [s]
epReachable na s = map snd (runNA na s [])

acceptNA :: (Alphabet l, Ord s) => NDetAut l s -> s -> [l] -> Bool
acceptNA na s0 w = any (\(ls,st) -> null ls && st `elem` naccept na) $ runNA na s0 w

ndfinalStates :: (Alphabet l, Ord s) => NDetAut l s -> s -> [l] -> [s]
ndfinalStates na s0 w = snd <$> runNA na s0 w
```

Finally, we implement algorithms to exhibit the behavioural equivalence between deterministic and non-deterministic automata. As discussed above, all DAs are of course NAs; but for every NA, we can also construct an DA with equivalent behaviours. The general idea is simple: We change the set of states to the set of *subsets* of the original non-determinisitic automaton. This way, we may code the non-deterministic behaviour in a deterministic way. The algorithm is inspired by [Fio10].

```
fromNA :: (Alphabet l, Ord s) => NDetAut l s -> DetAut l (Set.Set s)
fromNA nda = DA { states = Set.toList subsets
                , accept = Set.toList $ Set.filter acchelp subsets
                , delta = psetDelta }
  where subsets  = Set.powerSet $ Set.fromList $ nstates nda
        acchelp set = (not . Set.disjoint set . Set.fromList) acceptSingles
        acceptSingles = filter (\s -> (not . null) (intersect (epReachable nda s) (naccept nda)
            )) (nstates nda)
        psetDelta l s = Set.unions $ Set.map (onestep l) s
        onestep l s = Set.fromList $ result ++ concatMap (epReachable nda) result
          where result = (ndelta nda) (Just l) s

-- determinise NA, then run as as DA and check acceptance
dtdAccept :: (Alphabet l, Ord s) => NDetAut l s -> s -> [l] -> Bool
dtdAccept na st w = (run dtd initSt w) `elem` (accept dtd)
   where dtd = fromNA na
         initSt = Set.fromList (epReachable na st)
```

We test the claimed behavioural equivalence in Section 3.

1.3 Regular Expressions

```
module Regex where
```

```
import Alphabet ( Alphabet(..) )
```

In this section, we will define regular expressions, in the Kleene algebraic sense of the term. It's important to note that this version of regular expressions is different from those that are well known to programmers. For example, the language $\{a^n b a^n | n \in \mathbb{N}\}$ is well known to not be regular, and so not have a regular expression that represents it; meanwhile the programmer's regular expressions can encode this language rather easily.

The following serves as our definition of the Regex type. Note the distinction between `Empty` and `Epsilon`. The former is the empty language, while the latter is the empty string. Note also the type parameter `l`. This is so that we can use different input alphabets if we choose; see the `Alphabet` module for the definition of the `Alphabet` type class in Section 1.1.

```
data Regex l = Empty |
               Epsilon |
               L l |
               Alt (Regex l) (Regex l) |
               Seq (Regex l) (Regex l) |
               Star (Regex l)
  deriving (Eq, Show)
```

We also write a robust pretty printing function for `Regex`, with many hard coded cases so as to mimic the conventions of writing regular expressions as best we can. We then also include some quality-of-life functions, sequencing or alternating a list of regexes.

```
pPrintRgx :: Alphabet l => Regex l -> String
pPrintRgx Empty = "Em"
pPrintRgx Epsilon = "Ep"
pPrintRgx (L a) = pPrintLetter a
pPrintRgx (Alt (Seq r r') (Seq r'' r''')) = "(" ++ pPrintRgx (Seq r r') ++ ")+("
                                            ++ pPrintRgx (Seq r'' r''') ++ ")"
pPrintRgx (Alt (Seq r r') r'') = "(" ++ pPrintRgx (Seq r r') ++ ")" ++ "+" ++ pPrintRgx r''
pPrintRgx (Alt r'' (Seq r r')) = pPrintRgx r'' ++ "+" ++ "(" ++ pPrintRgx (Seq r r') ++ ")"
pPrintRgx (Alt r r') = pPrintRgx r ++ "+" ++ pPrintRgx r'
pPrintRgx (Seq (Alt r r') (Alt r'' r''')) = "(" ++ pPrintRgx (Alt r r') ++ ")("
                                            ++ pPrintRgx (Alt r'' r''') ++ ")"
pPrintRgx (Seq (Alt r r') r'') = "(" ++ pPrintRgx (Alt r r') ++ ")" ++ pPrintRgx r''
pPrintRgx (Seq r'' (Alt r r')) = pPrintRgx r'' ++ "(" ++ pPrintRgx (Alt r r') ++ ")"
pPrintRgx (Seq r (Star (L a))) = pPrintRgx r ++ "(" ++ pPrintLetter a ++ "*)"
pPrintRgx (Seq (Star (L a)) r) = "(" ++ pPrintLetter a ++ "*)" ++ pPrintRgx r
pPrintRgx (Seq r r') = pPrintRgx r ++ pPrintRgx r'
pPrintRgx (Star (L a)) = pPrintLetter a ++ "*"
pPrintRgx (Star r) = "(" ++ pPrintRgx r ++ ")*"

-- QoL functions for sequencing or alternating lists of regexes
seqList :: [Regex l] -> Regex l
seqList [] = Epsilon
seqList (l:ls) = Seq l $ seqList ls

altList :: [Regex l] -> Regex l
altList [] = Empty
altList (l:ls) = Alt l $ altList ls
```

It is outside of the scope of this project to implement an equivalence checker for regexes. Having said that, there are certain simplifications we can make to regular expressions that will help improve

readability and running time. For example:

$$\emptyset + r = r + \emptyset = r$$
$$\epsilon r = r\epsilon = r$$

...and more. The objective here is not to fully simplify the regular expression, but to implement easy simplifications to improve readability and runtime. We loop this function up to a fixed point.

```
simplifyRegex :: Eq l => Regex l -> Regex l
simplifyRegex regex | helper regex == regex = regex
                    | otherwise = helper regex where
 helper rx =
  case rx of
    Alt r4 (Seq r1 (Seq (Star r2) r3))
      | r1 == r2 && r3 == r4 -> Seq (Star (simplifyRegex r1)) (simplifyRegex r4)
    (Alt Empty r) -> simplifyRegex r
    (Alt r Empty) -> simplifyRegex r
    (Alt r r') | r == r' -> simplifyRegex r
    (Seq r Epsilon) -> simplifyRegex r
    (Seq Epsilon r) -> simplifyRegex r
    (Seq _ Empty) -> Empty
    (Seq Empty _) -> Empty
    (Star Empty) -> Epsilon
    (Star Epsilon) -> Epsilon
    (Star (Star r)) -> simplifyRegex $ Star r
    (Star (Alt r Epsilon)) -> simplifyRegex $ Star r
    (Star (Alt Epsilon r)) -> simplifyRegex $ Star r
    Alt r r' -> Alt (simplifyRegex r) (simplifyRegex r')
    Seq r r' -> Seq (simplifyRegex r) (simplifyRegex r')
    Star r -> Star (simplifyRegex r)
    x -> x
```

Now we need to set to the task of defining a semantics for these regular expressions. First, we will need a utility function for checking if initial sequences of the word satisfy part of the regex, specifically for the `Sequence` and `Star` cases.

This function takes a `Regex` and a word, and produces all splits of the word where the first part of the split satisfies the regex. For the word abc, with the regex $c^\star a^\star$, `initCheck` would output (ϵ, abc) and (a, bc).

```
initCheck :: Eq l => Regex l -> [l] -> [([l],[l])]
initCheck r w = filter (regexAccept r . fst) $ splits w where
  splits [] = [([],[])]
  splits (x:xs) = map (appFst x) (splits xs) ++ [([],x:xs)]
  appFst x (y,z) = (x:y,z)
```

Now we finally define the semantics for our regular expressions. We include special cases for sequencing with a single letter: because our general sequence case will be pretty inefficient, and this is a common use case that is quite fast.

In fact, both our sequencing and star operations are rather slow; this is because we need to examine all initial sequences of the word to see if they can match the regex. This is why `initCheck` returns the split of the word, rather than just the initial segment that matches: we need to then check the rest of the regex (the other operand in the case of `Seq`, or the regex again in the case of `Star`) on what remains of the string.

One potential optimisation we could make to this algorithm: calculate the range of string lengths a regex will accept, and compare this length to the input string, to find easier reject cases. We could pair this with splitting as we go, rather than up front.

```
regexAccept :: Eq l => Regex l -> [l] -> Bool
-- the empty language accepts no words
regexAccept Empty _    = False
-- if down to the empty string, only accept the empty word
regexAccept Epsilon [] = True
regexAccept Epsilon _  = False
-- if down to a single letter, only accept that letter
regexAccept (L _) []   = False
regexAccept (L l) [c]  = l == c
regexAccept (L _) _ = False
regexAccept (Alt r r') cs = regexAccept r cs || regexAccept r' cs
-- optimisations for simple sequences (one part is just a letter)
regexAccept (Seq (L _) _) [] = False
regexAccept (Seq _ (L _)) [] = False
regexAccept (Seq (L l) r) (c:cs) = l == c && regexAccept r cs
regexAccept (Seq r (L l)) cs = last cs == l && regexAccept r (init cs)
regexAccept (Seq Epsilon r) cs = regexAccept r cs
regexAccept (Seq r r') cs = any (regexAccept r' . snd) $ initCheck r cs
regexAccept (Star _) [] = True
regexAccept (Star r) cs =
  any (regexAccept (Star r) . snd) $ ignoreEmpty $ initCheck r cs
  where ignoreEmpty = if regexAccept r [] then init else id
```

In the general case of `Star`, similar to `Seq`, we want to find all initial segments of the word that satisfy the regular expression; but now we try to proceed using `Star r` again. There is an important subtlety, however: we want to avoid infinite looping, which may happen if the inner regular expression accepts the empty word.

Take for example the regex $(\epsilon + a)^\star$. This is equivalent to $a^\star$, of course, but introducing these kinds of simplifications to `simplifyRegex` significantly increases the complexity. If we're not careful, inputting the string bb with this regex will loop infinitely because we will continually find that (ϵ, bb) satisfies the $\epsilon + a$, and will loop back and forth between `regexAccept` and `initCheck` without making any forward progress in matching the word. To avoid this, we check if the inner regex accepts ϵ. If it does, we know it is redundant and we use `init` to drop the last accepting initial segment: namely (ϵ, w) for whatever input word w.

2 Kleene's Theorem

We have now defined (non)deterministic automata and regular expressions. Next, perhaps unsurprisingly, since these are well known to be two sides of the same coin, we encode a method to translate between them. Converting from a regex to a non-deterministic automaton is relatively straightforward, so we will begin with that. Second, we will describe Kleene's algorithm (a variation of the Floyd-Warshall Algorithm) in order to transform an automaton into a regex. Finally, we will note general problems with the above two steps as well as possible future methods to improve them.

```
module Kleene where

import Alphabet   ( Alphabet(..) )
```

```
import Automata   ( AutData(..), DetAut(..), NDetAut(..), encodeNA, trsOf )
import Regex      ( Regex(..), simplifyRegex, altList, seqList )
import Data.Maybe ( fromJust, isNothing )

import qualified Data.Map.Strict as Map

type RgxAutData l = (AutData l Int, Int)
```

2.1 Regular Expressions to Automata

Converting a regular expression into a non-deterministic automaton is both straightforward and (mostly) intuitive. Because regular expressions are defined inductively, we can associate an automaton with each base case, and then associate methods for combining automata for inductive cases. At a high level we can think of our algorithm as follows: first, the transition labels correspond to our alphabet; second, generate a very simple automaton for each atom (letter, epsilon, or empty string); then attach these automata together in a well-behaved way for each operator. Roughly, `Seq` corresponds to placing each automaton one after the other, `Alt` to placing them in parallel, and `Star` to folding the automaton into a circle. We will explain each of these operations more clearly in the appropriate section. As for any inductive construction we start with our base cases. Note these choices are not unique (there are many automata that accept no words) but we go with the simplest ones for each case:

`Empty` : a single, non-accepting, state.

`Epsilon` : a single, accepting, state.

`L a` : a non-accepting state, connected to an accepting one by a

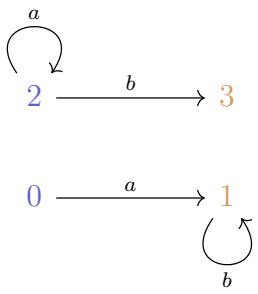
```
regToAut :: Regex l -> RgxAutData l
-- gives an integer automata and a starting state
regToAut Empty = (AD [1] [] [], 1)
regToAut Epsilon = (AD [1] [1] [], 1)
regToAut (L l) = (AD [1,2] [2] [(1, [(Just l,2)])], 1)
regToAut (Seq a b) = seqRegAut (regToAut a) (regToAut b)
regToAut (Alt a b) = altRegAut (regToAut a) (regToAut b)
regToAut (Star a)  = starRegAut $ regToAut a

-- convenience function for encoding result directly to an NA
fromReg :: (Alphabet l) => Regex l -> (NDetAut l Int, Int)
fromReg reg = (encodeNA ndata,st)
  where (ndata,st) = regToAut reg
```

An observant reader will have noticed this outputs an automata with `Int` states; this simply makes it easy to recurisvely generate new state labels in a well-bheaved way.

```
seqRegAut :: RgxAutData l -> RgxAutData l -> RgxAutData l
seqRegAut (aut1,s1) (aut2,s2) =
  (adata, 13*s1) where
  adata = AD
    ([ x*13 | x <- stateData aut1 ] ++ [ x*3 | x <- stateData aut2 ]) -- states
     [ 3*x | x <- acceptData aut2 ]                                   -- accepting
     (gluingSeq (aut1, s1) (aut2, s2))                                -- transitions
```

```
gluingSeq :: RgxAutData l -> RgxAutData l -> [(Int, [(Maybe l, Int)])]
gluingSeq (aut1, _) (aut2, s2) =  fstAut ++ mid ++ sndAut where
  fstAut = [mulAut 13 x (aut1 `trsOf` x) | x <- stateData aut1, x `notElem` acceptData aut1]
  mid = [appTuple (mulAut 13 x (aut1 `trsOf` x)) (Nothing,3*s2) | x <- acceptData aut1]
  sndAut = [mulAut 3 x (aut2 `trsOf` x) | x <- stateData aut2]
  mulAut n x trs = (x*n, multTuple n trs)
  appTuple (n,tr) t = (n,tr++[t])
```

This function takes two automata `aut1,aut2` and glues them together by adding epsilon transitions between the accepting states of `aut1` and the starting state of `aut2`. We need to add these epsilon transitions rather than merely identify the starting/ending in order to preserve transitions out of said states. For example, if we identify states 1 and 2 in the following two automata (blue states being intial, orange being accepting):



we would change the language from ab^*a^*b to $a(b^*a^*)^*b$. Thus, we add epsilon transitions.

Additionally, we multiply the states in the first automaton by 13 and states in the second automaton by 3. In the gluing and star operator we have to add new states (in order to prevent the gluing issue above). We always add a state labeled 1 for a starting state and 2 for an accepting state. Multiplication by prime numbers allows us to ensure that our new automaton *both* preserves the transition function of its component parts *and* has distinct state labels for every state. The primes are chosen arbitrarily, they simply ensure that each state has a unique label and allow us to understand which operations have been applied to which states for bugetsting-purposes.

```
altRegAut :: RgxAutData l -> RgxAutData l -> RgxAutData l
altRegAut (aut1, s1) (aut2, s2) =
  (adata, 1) where
  adata =  AD
    ([1,2] ++ [ x*5 | x<- stateData aut1 ] ++ [ x*7 | x<- stateData aut2 ])
    [2]
    (gluingAlt (aut1, s1) (aut2, s2))

gluingAlt :: RgxAutData l -> RgxAutData l -> [(Int, [(Maybe l, Int)])]
gluingAlt (aut1,s1) (aut2,s2) = start ++ fstAut ++endFstAut ++ sndAut ++ endSndAut where
  start = [(1, [(Nothing, s1*5), (Nothing, s2*7)])]
  fstAut = [mulAut 5 x (aut1 `trsOf` x) | x <- nonAccept aut1]
  sndAut = [mulAut 7 x (aut2 `trsOf` x) | x <- nonAccept aut2]
  endFstAut = [appSnd (Nothing,2) (mulAut 5 x (aut1 `trsOf` x)) | x <- acceptData aut1]
  endSndAut = [appSnd (Nothing,2) (mulAut 7 x (aut2 `trsOf` x)) | x <- acceptData aut2]
  mulAut n x trs = (x*n, multTuple n trs)
  nonAccept aut = [ x | x <- stateData aut, x `notElem` acceptData aut ]
  appSnd a (b,c) = (b,a:c)
```

Our construction for `Alt` is defined similarly to `Seq`. We add a new initial and acceptance state to ensure that our gluing preserves the appropriate transition functions. Lastly, we define our `Star` construction as follows (alongside some helper functions):

```
starRegAut :: RgxAutData l -> RgxAutData l
starRegAut (aut, s) =
  (adata, 1) where
  adata = AD
    (1:[x*11 | x<- stateData aut])
    [1]
    (gluingStar (aut, s))

gluingStar :: RgxAutData l -> [(Int, [(Maybe l, Int)])]
gluingStar (aut1, s1) = start ++ middle ++ end where
  start = [(1, [(Nothing, s1*11)])]
  middle = [(x*11, multTuple 11 (aut1 `trsOf` x))| x<-stateData aut1, x `notElem` acceptData
      aut1]
  end = [(a*11, (Nothing, 1) : multTuple 11 (aut1 `trsOf` a)) | a <- acceptData aut1]

multTuple :: Int -> [(a,Int)] -> [(a,Int)]
multTuple _ [] = []
multTuple n ((a,b):xs) = (a,n*b) : multTuple n xs

addTuple :: Int -> [(a,Int)] -> [(a,Int)]
addTuple _ [] = []
addTuple n ((a,b):xs) = (a,n+b) : multTuple n xs
```

For `Star` we add a single state which serves as both the initial state and an accept state. By connecting the beginning and ending of our starting automaton we create an abstract loop—corresponding to the operation `Star`.

This construction was relatively straightforward since by looking at what each operator in a regular expression means an automaton immediately suggests itself. The next algorithm, moving from automata to regular expressions, is far less intuitive, and encounters difficulties we will note in the final section. This complexity is due to the non-inductive/recursive definition of automata as opposed to regex.

2.2 Automata to Regular Expressions

Here, we implement an algorithim which take a non/deterministic automaton, a starting state, and outputs a corresponding regular expression. The algorithm we use, called Kleene's algorithm, allows us to impose a semi-recursive structure on an automaton which in turn allows us to extract a regular expression. First, we provide the implement of Kleene's algorithm (as well as some motivation and examples) before explaining how Kleene's algorithm can provide us with our desired conversion. We conclude with a brief overview of the helper functions we enlist throughout our implementation as well as a slightly different conversion (and why we opted with our method.)

Below, you will find our implementation of Kleene's algorithm; it take an automaton (whose states are labeled $[0..n]$ exactly), and three integers i, j, k (which correspond to states) and outputs a regular expression corresponding to the set of all paths from state i to state j without passing through states higher than k. The integer labels allow us to define a recursive function and it is easy to relabel states like this.

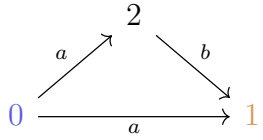
```
kleeneAlgo :: Eq l => AutData l Int -> Int -> Int -> Int -> Regex l
kleeneAlgo aut i j (-1) =
  if i==j
```

```
  then simplifyRegex (Alt Epsilon (altList [ maybe Epsilon L a | a <- successorSet aut i j]))
  else simplifyRegex (altList [ maybe Epsilon L a | a <- successorSet aut i j])
kleeneAlgo aut i j k =
  simplifyRegex (Alt (simplifyRegex $ kleeneAlgo aut i j (k-1))
                     (simplifyRegex $ seqList
                              [ simplifyRegex $ kleeneAlgo aut i k (k-1)
                              , simplifyRegex $ Star $ kleeneAlgo aut k k (k-1)
                              , simplifyRegex $ kleeneAlgo aut k j (k-1) ]))
```

Let us quickly dig in what this code actually means before moving on to an example. The algorithm successively removes states by decrementing k. At each step we remove the highest state and nicely add its associate transition labels to the remaining states. If `regToAut` worked by building up an automaton to follow a regular expression, Kleene's algorithm works by pulling a fully glued automaton apart step by step. When $k = -1$, we want to return a regex corresponding to the set of paths from i directly to j without stopping at any either state along the way. This is simply the set of transition labels which connect i to j (alongside `Epsilon` if $i = j$.) However, if $k > -1$, we need to remove the k'th state and shift the transition functions into and out of k amidst the rest of the automaton. First, we don't touch any of the paths which avoid k by including `kleeneAlgo aut` i j $(k-1)$. The remaining sequence can be viewed as: take any path to you want to k but stop *as soon as* you reach k for the first time; then, take any path from k to k as many times as you want; finally, take any path from k to j.

As we will see in the following example, this entire process can be though of as a single transition label encoding all of the data that used to be at k. By removing every state, we are left with a single arrow which corresponds to our desired regular expression. It is important to note that the algorithim does not *actually* change the autaumaton - rather this is a useful description of the recursive process the algorithim goes through.

In the following toy example, we apply the algorithim to this automata with intial state 0 and accepting state 1:



We apply the algorithim to 0 (the starting state) 1 (the accepting state) and 2 (the largest state.) By removing 2 from the state space (which is what decrementing k does), we get

$$(a\epsilon^* b) + a$$

which is exactly what the regular expression corresponding to this automaton is. Hopefully this very simple example has highlighted the motivation and process behind the algorithim.

With this broad motivation, we can know discuss how to implement the algorithm to provide our desired conversion:

```
autToReg :: Eq l => Ord s => (AutData l s, s)-> Regex l
autToReg (aut, s)= altList [kleeneAlgo intAut firstState a lastState | a <- acceptData intAut]
    where
  intAut = fst $ relabelAut (aut,s)
```

```
  firstState = relabelHelp aut s
  lastState = length (stateData intAut) - 1
```

This takes an automaton (and a starting state), transforms that automaton into one with the appropriate state labels and then applies the algorithm on the initial state and every accepting state. For a given accepting state a, (`kleeneAlgo aut` firstState a lastState) provides a regular expression corresponding to the paths from the initial state to a with no restrictions - we have set k to be higher than every state label. This is exactly what we were looking for, given our previous understanding of the algorithm itself.

Below we briefly describe the helper functions needed for this implementation as well as an alternate definition of autToReg, whose problems we will expound upon in the last section of this chapter. First, this function simply returns returns all the ways to directly move between two states for the base case of Kleene's algorithm.

```
-- takes automata data and states s1, s2. Outputs all the ways to get s2 from s1
successorSet :: Eq s => AutData l s -> s -> s -> [Maybe l]
successorSet aut s1 s2
  | isNothing (lookup s1 (transitionData aut) ) = [] -- if there are no successors
  | otherwise = map fst (filter (\w -> s2 == snd w) (fromJust $ lookup s1 (transitionData aut))
      )
```

Additionally, as mentioned, we need to convert an arbitrary automaton to one with well-behaved state labels in order to define Kleene's algorithm. These functions do so handily via the use of a dictionary.

```
-- states as integer makes everything easier. We use a dictionary to relabel in one sweep per
    say
relabelHelp :: Ord s => AutData l s -> s -> Int
relabelHelp aut s = fromJust (Map.lookup s (Map.fromList $ zip (stateData aut) [0 .. (length (
    stateData aut))]))

relabelAut :: Ord s => (AutData l s, s) -> RgxAutData l
relabelAut (aut, s1) = (AD [relabelHelp aut s | s <- stateData aut]
                           [relabelHelp aut s| s<- acceptData aut]
                           [(relabelHelp aut s, [ (a, relabelHelp aut b)| (a,b) <- aut `trsOf`
                               s] ) | s<- stateData aut]
                       , relabelHelp aut s1)
```

2.3 Issues with the Algorithms

The most prominent issue with this algoritihm is that it creates very complex regular expressions—each step adds a `Seq`, `Alt`, and `Star`. We have attempted to implement a few simplification throughout the algorithm, but it still outputs expressions that are horribly over-complex. For example,

$$\texttt{autToReg}\ (\texttt{wikiAutData}, 0) =$$
$$b + c + ((\epsilon + a)(a^*)(b + c)) +$$
$$((b + c + ((\epsilon + a)(a^*)(b + c)))(\epsilon + b + ((a + c)(a^*)(b + c)))^*(\epsilon + b + ((a + c)(a^*)(b + c))))$$

which, upon manual reduction, is equivalent to

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

However, reduction of regular expressions is NP-hard, and so we have simply tried to encode a few, computationally quick, simplifications as noted in Section 1.3. This is most pressing when we convert back into an automaton, since each new operator corresponds to an additional structural level in the automaton and an additional complication when running words on automata. There are several ways this could be improved. Perhaps most straightforward would be to further improve our regular expression simplifcation—or change the type definition to more easily manage commutativity. Another possibility would be to implement other algorithms aside from Kleene's which might be better-fit for converting certain automata, comparing results across them. Finally, we could implement automaton-minimisation.

3 Testing

3.1 Examples

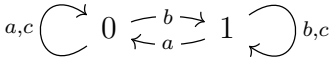
In this section we define examples of (non)deterministic automata and regular expressions, and run tests to verify correctness of our algorithms. More concretely, we will test:

- Basic semantic layers for (non)deterministic automata and regular expressions
- Converting between deterministic and non-deterministic automata (from Section 1.2)
- Converting between (non)deterministic automata and regular expressions (from Section 2)

We begin by including the relevant modules.

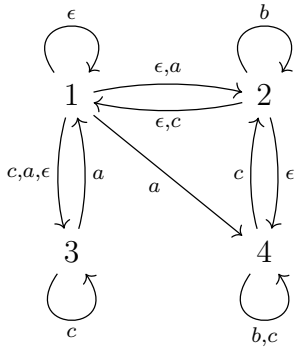
```
module Examples where

import Alphabet ( Letter(..) )
import Automata
    ( AutData(AD), DetAut, NDetAut,
      encodeDA, encodeNA, run, acceptDA )
import Regex ( Regex(..) )
import Data.Maybe ( fromJust )
```

Now we give some examples of automata. Our first is the following deterministic automaton, taken from the Wikipedia page for Kleene's algorithm:



```
wikiAutData :: AutData Letter Int
wikiAutData = AD [0,1]   -- the states
                 [1]     -- the accepting state
                 [(0, [(Just A, 0) ,(Just B, 1) ,(Just C, 0)]) -- transitions
                 ,(1, [(Just A, 0) ,(Just B, 1) ,(Just C, 1)])]

wikiDA :: DetAut Letter Int
wikiDA = fromJust $ encodeDA wikiAutData
```

We also consider this example of a non-deterministic automaton. Note how the data is not formatted properly: transitions from state 1 are not all listed together, and there are unnecessary epsilon loops that will make our implmementation of NA semantics loop. When encoding this automaton, we will fix these formatting issues: see Section 1.2 for details.




```
myNAutData :: AutData Letter Int
myNAutData = AD [1,2,3,4]          -- the states
                [4]                -- accepting states
                [(1,[(Nothing,2),(Just C,3)])
                ,(2,[(Nothing,1),(Nothing,4),(Just B,2),(Just C,1)])
                -- want to merge these with above
                ,(1,[(Just A,2),(Just A,3),(Nothing,1),(Nothing,3),(Just A,4)])
                ,(3,[(Just A,1),(Just C,3)])
                ,(4,[(Just B,4),(Just C,4)])]

myNDA :: NDetAut Letter Int
myNDA = encodeNA myNAutData -- this automaton, encoded
```

Here is an example for regular expressions:

```
exampleRegex :: Regex Letter
exampleRegex = Seq (Star (Alt (L A) (L B))) (Alt (Star (L A)) (Alt Epsilon (L B)))
-- pPrints as (a+b)*(a*+Ep+b)
```

3.2 QuickCheck

We now use the library QuickCheck to randomly generate input for our functions and test some properties.

```
module Main where

import Test.Hspec ( hspec, describe )
import Test.Hspec.QuickCheck ( prop )
import Automata ( acceptDA, decode, fromDA, acceptNA, dtdAccept )
import Regex ( regexAccept )
import Kleene ( autToReg, fromReg )
```

```
import Examples ( exampleRegex, myNDA, wikiDA )
```

We have tested behavioural equivalence using randomly generated words. We use a regex, a DA, and an NA as starting points, convert them using one of our functions, and compare the semantics of the original with the new version.

```
main :: IO ()
main = hspec $ do
  describe "Examples" $ do
    prop "NA and transfer to DA should give the same result" $
      \input -> acceptNA myNDA 1 input == dtdAccept myNDA 1 input
    prop "reg to NA should give the same result" $
      \input -> regexAccept exampleRegex input == uncurry acceptNA (fromReg exampleRegex) input
    prop "DA to reg should give the same result" $
      \input -> acceptDA wikiDA 0 input == regexAccept (autToReg ((decode . fromDA) wikiDA, 0))
          input
    prop "NA to reg should give the same result" $
      \input -> acceptNA myNDA 1 input == regexAccept (autToReg (decode myNDA, 1)) input
```

The result is recorded below. In the beta version of this report, we had to restrict the input size for some cases, but optimisations to the NA semantics made since then have made the limit unnecessary; and we managed to cut our benchmark time in half, too. The main optimisation was to move from tracking active states with a list to using a `Set`, cutting down on redundant work, and fixing a bug that allowd for infinite looping.

```
Examples
  NA and transfer to DA should give the same result [\/]
    +++ OK, passed 100 tests.
  reg to NA should give the same result [\/]
    +++ OK, passed 100 tests.
  DA to reg should give the same result [\/]
    +++ OK, passed 100 tests.
  NA to reg should give the same result [\/]
    +++ OK, passed 100 tests.

Finished in 3.6397 seconds
4 examples, 0 failures
```

References

[Cox07] Russ Cox. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...). `https://swtch.com/~rsc/regexp/regexp1.html`, 2007.

[Fio10] Marcelo Fiore. *Lecture Notes on Regular Languages and Finite Automata*. Accessible at `https://www.cl.cam.ac.uk/teaching/1011/RLFA/LectureNotes.pdf`, 2010.