DFA, NFA and regular expression conversions in Haskell

Roy Overbeek

Autumn 2016

Abstract

We have developed Haskell code for conversions between deterministic finite state automata (DFAs), non-deterministic finite state automata (NFAs) and regular expressions (regexes). The algorithms will eventually be used for generating random exercise instances (and automatically verifying solutions) for the Vrije Universiteit BSc Computer Science course Automata and Complexity.

Contents

1 Introduction

BSc Computer Science students at the Vrije Universiteit are required to take the (self-descriptive) course Automata and Complexity.[1] The textbook used in the course is *An Introduction to Formal Languages and Automata* by Peter Linz [Lin06]. Currently, there is no environment for students in which they can practice the algorithms related to automata and regular expressions.

I have been appointed to implement random task generation in Scala for such an environment. For the FSA project, I have therefore decided to do some of the prototyping in Haskell. Apart from the basic functionalities of DFAs, NFAs and regular expressions, I have managed to implement the following algorithms:

- NFA to DFA conversion;
- DFA minimization;
- Regex to NFA conversion.

In addition, I have partially implemented regex to NFA conversion. It is unfinished and undiscussed due to time constraints.

Note that for all these algorithms, Linz's exposition is intentionally followed closely (another option could be to use the most efficient algorithms in the backend). This way it is easier to provide students with relevant feedback. Also, if some future tasks ask students to compute steps of an algorithm, then the code required for such task generation is already contained in the code. Optimizations can wait until they are necessary.

2 Preliminaries

In this section we summarize all basic concepts, and we show their implementation in Haskell.

2.1 Languages

An *alphabet* Σ is a set of *symbols*. A *string* on Σ is a finite sequence of symbols from Σ. The empty string is denoted ε. The *concatenation of strings* w and v is denoted wv. Σ^* (*Kleene star*) is the set containing all strings over Σ, while $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.

[1] The course description can be found at: `http://www.vu.nl/nl/studiegids/2016-2017/bachelor/c-f/computer-science/index.aspx?view=module&origin=50049131&id=50052470`

Any subset L of Σ^* is called a *language* over Σ. The *concatenation of languages* L_1 and L_2 is defined by the set $\{xy : x \in L_1, y \in L_2\}$. The language L^n denotes the set L concatenated with itself n times, with special cases $L^0 = \{\varepsilon\}$ and $L^1 = L$. Finally, $L^* = \{L^n \mid n \in \mathbb{N}\}$ and $L^+ = \{L^n \mid n \in \mathbb{N}^+\}$.

In Haskell, we use `Char`s for symbols. And in general, we often use lists instead of sets for purposes of pattern matching. The main disadvantage of this is that one must make sure that duplicate elements are never introduced.

Thus the basic types are as follows:

```
type Symbol = Char
type Alphabet = [Symbol]
type Language = [String]

-- epsilon itself isn't rendered by show, so @ will have to do
eps :: Symbol
eps = '@'
```

And the language operations are implemented by:

```
kleeneStar :: Alphabet -> Language
kleeneStar alphabet = "" : extend [""]
  where extend prev = let curr = [s:x | s <- alphabet, x <- prev]
                      in curr ++ extend curr

kleenePlus :: Alphabet -> Language
kleenePlus = tail . kleeneStar

concatenate :: Language -> Language -> Language
concatenate l1 l2 = [x ++ y | x <- l1, y <- l2]

ln :: Language -> Int -> Language
ln l 0 = [""]
ln l 1 = l
ln l n = concatenate l (ln l (pred n))

starClosure :: Language -> Language
starClosure l = concat [ln l n | n <- [0..]]

posClosure :: Language -> Language
posClosure l = tail \$ starClosure l
```

2.2 Automata

A *DFA* is a quintuple $(Q, \Sigma, \delta, q_0, F)$ with Q a finite set of *states*, Σ a finite set of *symbols* (called the *input alphabet*), $\delta : Q \times \Sigma \to Q$ a total *transition function*, $q_0 \in Q$ the *initial state* and $F \subseteq Q$ a set of *final states*. An *NFA* is defined identically, except that $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$, with $\varepsilon \notin \Sigma$, and δ a partial function.

In Haskell, DFAs and NFAs are represented as follows:

```
newtype DFA a = DFA ([a], Alphabet, Map.Map (a,Symbol) a  , a, [a])
newtype NFA a = NFA ([a], Alphabet, Map.Map (a,Symbol) [a], a, [a])
```

First, note that a type variable is used for the states. This is practical, since a

convenient canonical type for states is `Int`, while we want to use states of type `[Int]` when converting an NFA to a DFA. Second, note that a `Map` (from library `Data.Map`) is used for the transition function, so that lookup and insertion operations are in $O(log\ n)$. The use of `Map` requires that that the type variable a is a member of typeclass `Ord`. Finally, we use `newtype` over `data` since `newtype` results in a faster implementation, and we do not need more than one constructor.

DFAs, NFAs and other automata that may be implemented later (such as pushdown automata) have a lot of functionality in common. For each of them we can ask what states we can reach given some starting state (or states) and a symbol or a string, whether it accepts a word, and which language it generates. Since automata are unwieldy tuples, it is also convenient to use extractor functions. Thus it makes sense to make an `Automata` typeclass:

```
class (Ord state, Ord mapsTo) => Automaton a state mapsTo | a -> state, a ->
    mapsTo where
  states    :: a -> [state]
  sigma     :: a -> Alphabet
  delta     :: a -> Map.Map (state,Symbol) mapsTo
  initial   :: a -> state
  final     :: a -> [state]

  valid     :: a -> Bool
  consume   :: a -> [state] -> Symbol -> [state]
  reachable :: a -> [state] -> String -> [state]
  accepts   :: a -> String -> Bool

  language  :: a -> Language
  language fa = filter (accepts fa) (kleeneStar (sigma fa))

  accessibleStates :: a -> [state]
  accessibleStates fa = cover (\x -> concat [reachable fa x (symbol:"") | symbol
       <- sigma fa]) [initial fa]
```

where `cover` is defined by:

```
cover :: Eq a => ([a] -> [a]) -> [a] -> [a]
cover f x = let y = nub (x ++ f x) in if length x == length y then x else cover
    f y
```

In standard Haskell one can use only one parameter per typeclass. Here, we use three, using the Multi-Parameter Typeclasses language extension. For safety, we also use the Functional Dependencies extension to put contraints on parameters `state` and `mapsTo`: given `a`, `state` and `mapsTo` are uniquely identified. More particularly, a DFA over `a` implies that `state` and `mapsTo` are both of type `a`, while an NFA over `a` implies that `state` is of type `a` and `mapsTo` of type `[a]`.

Because we use nested variables in our instantiations of the `automaton` class, we also need to use the Flexible Instances language extension. All of these extensions are enabled through declarations at the head of the file:

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE FlexibleInstances #-}
```

We do not discuss the DFA and NFA instantiations here, and hope they are sufficiently

self-documenting.

2.3 Regular expressions

A *regular expression* over an alphabet Σ is defined inductively as follows:

1. $\varnothing$, ε and any $a \in \Sigma$ are all (primitive) regular expressions;
2. If r is a regular expression, so is r^*;
3. If r_1 and r_2 are regular expressions, so are $(r_1 + r_2)$ and $(r_1 \cdot r_2)$.

Note that in writing we use overloading expressions: $\varnothing$ can denote either the empty set or a primitive regular expression. From context it should be clear which is meant. In Haskell there is no such overloading.

Like DFAs and NFAs, regular languages are used to describe regular languages. The language $L(\cdot)$ associated with a regular expression r is as follows:

1. $L(\varnothing) = \varnothing$;
2. $L(\varepsilon) = \{\varepsilon\}$;
3. $L(a) = \{a\}$ (for $a \in \Sigma$);
4. $L((r_1 + r_2)) = L(r_1) \cup L(r_2)$;
5. $L((r_1 \cdot r_2)) = L(r_1)L(r_2)$;
6. $L(r_1^*) = (L(r_1))^*$.

The corresponding Haskell code is as follows:

```
data Regex = Empty | Eps | S Symbol |
             Plus Regex Regex | Dot Regex Regex | Star Regex
  deriving (Eq, Ord)

lang :: Regex -> Language
lang Empty        = []
lang Eps          = [""]
lang (S c)        = [c:""]
lang (Plus r1 r2) = lang r1 ++ lang r2
lang (Dot r1 r2)  = concatenate (lang r1) (lang r2)
lang (Star r1)    = starClosure (lang r1)
```

Of course, the implementation can be refined later by adding a string parser (it is currently very cumbersome to write a regex) and priority rules for operators.

3 Algorithms

In this section we look at the algorithms mentioned in the introduction: NFA to DFA conversion, DFA minimization and regex to NFA conversion.

3.1 NFA to DFA

Any NFA $M_N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ can be converted to a DFA $M_D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$, where $Q_D \subseteq \mathbb{P}(Q_N)$. The procedure outlined in Linz is roughly as follows: initialize a graph G_D containing just the initial vertex $\{q_0\}$. Then, as long as some transition is missing for some $\{q_i, q_j, \ldots, q_k\}$ in G_D and $a \in \Sigma$ (recall that δ_D must be total), compute which states $q_l, q_m, \ldots, q_n$ are reachable in M_N, upon consumption of a while at states $q_i, q_j, \ldots, q_k$. Add an a-transition from $\{q_i, q_j, \ldots, q_k\}$ to $\{q_l, q_m, \ldots, q_n\}$ (add $\{q_l, q_m, \ldots, q_n\}$ as a vertex if it does not yet exist). When this routine finishes, any state of G_D containing a state of F_N is marked final. If M_N accepts ε, then $\{q_0\}$ is also marked final.

In Haskell, this algorithm is implemented as follows:

```
getMissingTransition :: (Ord a) => DFA a -> Maybe (a,Symbol)
getMissingTransition dfa =
  let missing = reqTransitions \\ Map.keys (delta dfa)
  in  if null missing then Nothing else Just (head missing)
  where reqTransitions = [(s,x) | s <- states dfa, x <- sigma dfa]

nfa2dfa :: (Ord a) => NFA a -> DFA [a]
nfa2dfa nfa = complete (DFA ([[initial nfa]], sigma nfa, Map.empty, [initial nfa
    ], []))
  where complete dfa@(DFA (q,s,d,i,[])) =
          let toAdd = getMissingTransition dfa
          in case toAdd of
            Just (state,symbol) ->
              let target = consumeClosure nfa state symbol
              in  let existing = filter (coextensive target) q
                  in  complete \$ DFA \$
                        if null existing
                        then (target:q, s, Map.insert (state,symbol) target d, i
                            , [])
                        else (q, s, Map.insert (state,symbol) (head existing) d,
                             i, [])
            Nothing ->
              let f = filter (any (`elem` final nfa)) q
              in  DFA \$ if any (`elem` final nfa) (epsilonReachable nfa i)
                        then (q, s, d, i, i:f)
                        else (q, s, d, i, f  )
```

3.2 DFA minimization

To minimize a DFA M, one first removes all inaccessible states. Then one marks all *distinguishable* states. Two states s and t are distinguishable if there exists a string w such that M accepts w starting at s, while it does not accept w starting at t (or

vice versa). one can compute distinguishable states as follows:

1. For all states p and q, mark the pair (p, q) distinguishable if $p \in F$ and $q \notin F$ or vice versa. (On the empty string, these states give different outcomes.)
2. Repeat until no new pairs are marked: for all pairs (p, q) and all $a \in \Sigma$, compute $\delta(p, a)$ and $\delta(q, a)$. If $(\delta(p, a), \delta(q, a))$ is distinguishable, then mark (p, q) as distinguishable.

Once all distinguishable states are found, partition Q under the indistinguishability relation. These are the states of the minimized DFA M'. For all transitions $\delta(p, a) = q$ in M, find the corresponding equivalence classes p_E and q_E, and add the transition $\delta(p_E, a) = q_E$ to M'. The initial state of M' is the state which contains the initial state of M. The final states of M' are those states which contain some final state of M.

```
removeInaccessible :: Ord a => DFA a -> DFA a
removeInaccessible dfa@(DFA (q,s,d,i,f)) =
  DFA (accessible,s,d',i,f')
  where accessible = accessibleStates dfa
        f' = f `intersect` accessible
        d' = Map.fromList \$ filter (\((x,y),z) -> x `elem` accessible && z `
            elem` accessible) (Map.toList d)

mark :: Ord a => DFA a -> [(a,a)]
mark dfa@(DFA (q,s,d,i,f)) = cover ext distinguishable
  where ext marked = [(x,y) | (x,y) <- cart q q, sym <- s,
                              (head (consume dfa [x] sym), head (consume dfa [y]
                                   sym)) `elem` marked]
        distinguishable = cart (q \\ f) f ++ cart f (q \\ f)

equivClasses dfa = nub \$ generateEquiv (states dfa)
  where indist = cart (states dfa) (states dfa) \\ mark dfa
        generateEquiv = map (\x -> map snd (filter ((==) x . fst) indist))

minimize :: Ord a => DFA a -> DFA [a]
minimize dfa@(DFA (q,s,d,i,f)) = DFA (q',s,d',i',f')
  where q' = equivClasses dfa
        i' = head \$ filter (i `elem`) q'
        d' = Map.fromList \$ nub \$ buildTable \$ Map.toList d
        buildTable [] = []
        buildTable (((a,b),c) : xs) = ((head (filter (a `elem`) q'), b), head (
            filter (c `elem`) q')) : buildTable xs
        f' = filter (not . null . intersect f) q'
```

3.3 Regex to NFA

Linz explains regex to NFA conversion according to Thompson's construction, see `https://en.wikipedia.org/wiki/Thompson's_construction#Rules` (unfortunately, I don't have time to render these images myself). The only difference is that Linz puts one ε-transition in the construction for r^* on the outer nodes instead of the inner nodes, but this is inconsequential.

In Linz and on Wikipedia, states are anonymous. In our case states need to be relabelled to prevent conflicts. The code is as follows:

```
regex2nfa :: Regex -> NFA Int
regex2nfa Empty        = NFA ([0,1], [],  Map.empty                    , 0,
    [1])
regex2nfa Eps          = NFA ([0,1], [],  Map.fromList [(0, eps) ~> [1]], 0,
    [1])
regex2nfa (S c)        = NFA ([0,1], [c], Map.fromList [(0,  c ) ~> [1]], 0,
    [1])
regex2nfa (Plus r1 r2) =
  NFA (q,s,d,i,[f])
    where nfaR1 = regex2nfa r1
          nfaR2 = let min = maximum (states nfaR1) + 1
                  in  relabelNFA (regex2nfa r2) [min..]
          max = maximum (states nfaR2)
          i = max+1
          f = max+2
          q = [i,f] ++ states nfaR1 ++ states nfaR2
          s = nub $ sigma nfaR1 ++ sigma nfaR2
          finalTargetsR1 = let x = Map.lookup (head (final nfaR1), eps) (delta
              nfaR1)
                           in  fromMaybe [] x
          finalTargetsR2 = let x = Map.lookup (head (final nfaR2), eps) (delta
              nfaR2)
                           in  fromMaybe [] x
          d = let existing = Map.union (delta nfaR1) (delta nfaR2)
              in Map.fromList $
                  Map.toList existing ++
                  [(i, eps)                  ~> [initial nfaR1, initial nfaR2],
                   (head (final nfaR1), eps) ~> (f:finalTargetsR1),
                   (head (final nfaR2), eps) ~> (f:finalTargetsR2)
                  ]
regex2nfa (Dot r1 r2) =
  NFA (q,s,d,i,[f])
    where nfaR1 = regex2nfa r1
          nfaR2 = let min = maximum (states nfaR1) + 1
                  in relabelNFA (regex2nfa r2) [min..]
          max = maximum (states nfaR2)
          i = max+1
          f = max+2
          q = [i,f] ++ states nfaR1 ++ states nfaR2
          s = nub $ sigma nfaR1 ++ sigma nfaR2
          finalTargetsR1 = let x = Map.lookup (head (final nfaR1), eps) (delta
              nfaR1)
                           in  fromMaybe [] x
          finalTargetsR2 = let x = Map.lookup (head (final nfaR2), eps) (delta
              nfaR2)
                           in  fromMaybe [] x
          d = let existing = Map.union (delta nfaR1) (delta nfaR2)
              in Map.fromList $
                   Map.toList existing ++
                   [(i, eps)                  ~> [initial nfaR1],
                    (head (final nfaR1), eps) ~> (initial nfaR2:finalTargetsR1),
                    (head (final nfaR2), eps) ~> (f:finalTargetsR2)
                   ]
regex2nfa (Star r1) =
  NFA (q,s,d,i,[f])
    where nfaR1 = regex2nfa r1
          max = maximum (states nfaR1)
          i = max+1
          f = max+2
          q = [i,f] ++ states nfaR1
          s = sigma nfaR1
          finalTargetsR1 = let x = Map.lookup (head (final nfaR1), eps) (delta
              nfaR1)
                           in  fromMaybe [] x
          d = Map.fromList \$
              Map.toList (delta nfaR1) ++
                  [(i, eps)                  ~> [initial nfaR1, f],
```

```
                  (head (final nfaR1), eps) ~> (f:finalTargetsR1),
                  (f, eps)                  ~> [i]
                ]
```

Using this algorithm results in a state explosion. But we can easily convert to a minimal DFA instead:

```
regex2dfa :: Regex -> DFA Int
regex2dfa = (\x -> relabelDFA x [0..]) . minimize . nfa2dfa . regex2nfa
```

4 Discussion

There is a lot of room for improvement in the existing code. For instance, generating the language of some automaton boils down to checking all strings for acceptance: shortcuts could be used. In addition, some functions, such as `regex2nfa`, could be written to be more concise.

References

[Lin06] Peter Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett Publishers, 4th edition, 2006.