Haskelleene

a very Haskell implmentation of automata, regular expressions, and Kleene's algorithm

Liam Chung, Brendan Dufty, Lingyuan Ye

31st May, 2024

Table of Contents

Table of Contents

What is an automaton?

An basic version of a state machine. It takes inputs from some *alphabet*, moving between *states* that may or may not *accept*.

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

data NDetAut l s = NA { nstates :: [s]
                      , naccept :: [s]
                      , ndelta :: Maybe l -> s -> [s] }
```

What is an automaton?

An basic version of a state machine. It takes inputs from some *alphabet*, moving between *states* that may or may not *accept*.

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

data NDetAut l s = NA { nstates :: [s]
                      , naccept :: [s]
                      , ndelta :: Maybe l -> s -> [s] }
```

- a state in a deterministic automaton accepts a word if that words leads to an accepting state.

What is an automaton?

An basic version of a state machine. It takes inputs from some *alphabet*, moving between *states* that may or may not *accept*.

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

data NDetAut l s = NA { nstates :: [s]
                      , naccept :: [s]
                      , ndelta :: Maybe l -> s -> [s] }
```

- a state in a deterministic automaton accepts a word if that words leads to an accepting state.
- a state in a non-deterministic automaton accepts a word if *there exists a path* to an accepting state.

Examples

A deterministic automaton:

$$\begin{tikzcd} 1 \arrow[loop left, "b"] \arrow[r, bend left, "a"] & 2^{\star} \arrow[l, bend left, "b"] \arrow[loop right, "a"] \end{tikzcd}$$

Examples

A deterministic automaton:

$$\begin{tikzcd} 1 \arrow[loop left, "b"] \arrow[r, bend left, "a"] & 2^\star \arrow[l, bend left, "b"] \arrow[loop right, "a"] \end{tikzcd}$$

A non-deterministic one:

$$\begin{tikzcd} 1 \arrow[r, shift left, "\epsilon"] \arrow[d, "b"] & 2 \arrow[l, shift left, "c"] \arrow[dl, "a"] \arrow[d, "\epsilon"] \arrow[loop above, "b"] \\ 3 & 4^\star \end{tikzcd}$$

Algorithm for Running DAs

Given an automaton, a starting state, and an input word...

```
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
```

Algorithm for Running DAs

Given an automaton, a starting state, and an input word...

```
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
```

...it's that simple! Mostly due to how much work we put into encoding automata.

Algorithm for Running DAs

Given an automaton, a starting state, and an input word...

```
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
```

...it's that simple! Mostly due to how much work we put into encoding automata.

Unfortunately, running NAs is less simple.

Algorithm for Running NAs

General Idea

Searching paths in a finite graph in general requires a lot of computational resources. However, we do not need to output the whole path, thus we only keep track of how states transform.

Algorithm for Running NAs

General Idea

Searching paths in a finite graph in general requires a lot of computational resources. However, we do not need to output the whole path, thus we only keep track of how states transform.

Say we run the input *ba* on the previous example:

$$\begin{tikzcd}
1 \arrow[r, shift left, "\epsilon"] \arrow[d, "b"] & 2 \arrow[l, shift left, "c"] \arrow[dl, "a"] \arrow[d, "\epsilon"] \arrow[loop above, "b"] \\
3 & 4
\end{tikzcd}$$

Active states: $([ba], 1)$.

Algorithm for Running NAs

General Idea

Searching paths in a finite graph in general requires a lot of computational resources. However, we do not need to output the whole path, thus we only keep track of how states transform.

Say we run the input *ba* on the previous example:

$$\begin{tikzcd}
1 \arrow[r, shift left, "\epsilon"] \arrow[d, "b"] & 2 \arrow[l, shift left, "c"] \arrow[loop above, "b"] \arrow[dl, "a"] \arrow[d, "\epsilon"] \\
3 & 4
\end{tikzcd}$$

Active states: $([a], 3), ([ba], 2)$.

Algorithm for Running NAs

General Idea

Searching paths in a finite graph in general requires a lot of computational resources. However, we do not need to output the whole path, thus we only keep track of how states transform.

Say we run the input *ba* on the previous example:

$$
\begin{tikzcd}
1 \arrow[r, shift left, "\epsilon"] \arrow[d, "b"] & 2 \arrow[l, shift left, "c"] \arrow[dl, "a"] \arrow[d, "\epsilon"] \arrow[loop above, "b"] \\
3 & 4
\end{tikzcd}
$$

Active states: $([a], 3), ([a], 2), ([ba], 4)$.

Algorithm for Running NAs

General Idea

Searching paths in a finite graph in general requires a lot of computational resources. However, we do not need to output the whole path, thus we only keep track of how states transform.

Say we run the input *bca* on the previous example:

$$\begin{tikzcd}
1 \arrow[r, shift left, "\epsilon"] \arrow[d, "b"] & 2 \arrow[l, shift left, "c"] \arrow[loop above, "b"] \arrow[dl, "a"'] \arrow[d, "\epsilon"] \\
3 & 4
\end{tikzcd}$$

Active states: $([a], 3), ([], 3), ([a], 4), ([ba], 4)$.

Haskell Implementation of Semantics for NA

The function `runNA` is defined as follows:

```
runNA :: (Alphabet l, Ord s) =>
         NDetAut l s  -> s -> [l] -> [([l], s)]
runNA na st input =
  case input of
    [] -> ([],) <$> epReachable (ndelta na) st
    (w:ws) -> concatMap (\s -> runNA na s input) nsucc ++
              case wsucc of
                [] -> [(input,st)]
                ls -> concatMap (\s -> runNA na s ws) ls
      where wsucc = ndelta na (Just w) st
    where   nsucc = ndelta na Nothing  st
```

Here the function `epReachable` calculates states reachable from the current one via ϵ-transitions.

Equivalence between DA and NA

Evidently, any DA is a NA. On the other hand, we can simulate running NA deterministically, basically via the same idea as `runNA`:

- States are subsets of states of a NA.
- A subset is accepting iff it contains some accepting state.
- Under an input I, a subset transforms to those states reachable from some state via I (with ϵ-transitions).

Haskell Implementation

```
fromNA :: (Alphabet l, Ord s) =>
          NDetAut l s -> DetAut l (Set.Set s)
fromNA nda = DA { states = Set.toList dasts
                , accept = Set.toList $ Set.filter
                    acchelp dasts
                , delta = fromTransNA ntrans
                }
  where ndasts = nstates nda
        dasts  = Set.powerSet $ Set.fromList ndasts
        ndaacc = naccept nda
        acchelp set = not $ Set.disjoint set
                          $ Set.fromList ndaacc
        ntrans = ndelta nda

fromTransNA :: (Alphabet l, Ord s) =>
               (Maybe l -> s -> [s]) -> l -> Set.Set s ->
                    Set.Set s
fromTransNA ntrans sym set = result
  where starts = listUnions (epReachable ntrans) set
        step = listUnions (ntrans $ Just sym) starts
        result = listUnions (epReachable ntrans) step
        listUnions f input = Set.unions $ Set.map Set.
            fromList $ Set.map f input
```

Table of Contents

What is a regular expression?

A "finite representation" of a potentially infinite language:

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

N.B. not the same as the commonly known, "programmer's" regular expression!

Implementing semantics

How to check if a word is in the language described by a `Regex`?

```
regexAccept :: Eq l => Regex l -> [l] -> Bool
-- the empty language accepts no words
regexAccept Empty _      = False
-- if down to empty string, only accept 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') w =
                  regexAccept r w || regexAccept r' w
...
```

These cases are not so bad!

Seq cases

First, we will need the following:

```
initCheck :: Eq l => Regex l -> [l] -> [([l],[l])]
initCheck r w = filter (regexAccept r . fst) (splits w)
```

...which finds all initial segments of the word matching `r`.

```
...
regexAccept (Seq r r') w = any (regexAccept r' . snd)
                               (initCheck r w)
...
```

For every valid split using `r`, try `r'` on the rest.

Star case

Similar for Star, with one caveat:

```
...
regexAccept (Star r) w = any (regexAccept (Star r) . snd)
                             (ignoreE (initCheck r w))
  where ignoreE = if (regexAccept r []) then init else id
```

we need to avoid infinite looping by accepting the empty split!

Table of Contents

The Theorem

Theorem

Every regular expression corresponds to a non-deterministic automaton and vice-versa

Regular Expression to Automaton

Regular expression are built inductively

↓

Define a way to build autmata inductively

↓

Base Cases

↓

Each regex constructor $\rightarrow$ a way to nicely *glue* automata together

Base Cases

`Empty:` A single non-accepting state.

`Epsilon:` A single accepting state.

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

Inductive Vibes

Given two (one) automata and a regular expression constructor, make a new automaton

$\boxed{\texttt{Seq:}}$ Attach both automata end to end.

$\boxed{\texttt{Alt}}$ Attach both automata in parralel (with new initial and final states).

$\boxed{\texttt{Star}}$ Fold the automaton in a circle around a new initial/final state.

Interesting Implementation

New states $\rightarrow$ consistent labeling $\rightarrow$ primes!

Epsilon transitions guarantee transition function fidelity (Outputs a NA).

Automata to Regular Expression

Too much to give an understandable dive into the algorithm

Summary of algorithm

Highlight some implementation

Problems and Future Work

Kleene's Algorithim

Also known as State-Elimination

Goal: Recursively deconstruct an automata to get a regex

Easy to implement - Haskell likes recursion!

Quick Blackboard Example

Implementation

Need to relabel an automata onto integers

```
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 `transOf` s] ) | s<- stateData aut]
,relabelHelp aut s1)
```

Problems

Outputs highly unsimplified regex - hard to simplify

Explodes when converting aut $\rightarrow$regex$\rightarrow$aut or vice-versa

VERY SLOW

Future Work

Construct minimal DA from regex to prove algorithm validates Kleene's Theorem

Further simplify regular expressions (commutativity?)

Test other algorithms and compare with Kleene's