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

Introduction

"Automata are pretty cool."

\- Liam Chung

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] }
```

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

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] }
```

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

How do we encode `delta`**?**

Automaton data: `AutData`

A simpler representation of our data:

```
type TDict l s = [(s, [(Maybe l, s)])]
data AutData l s = AD { stateData :: [s]
                      , acceptData :: [s]
                      , transitionData :: TDict l s }
```

Then, encode this data into our nicer types.

Automaton data: AutData

A simpler representation of our data:

```
type TDict l s = [(s, [(Maybe l, s)])]
data AutData l s = AD { stateData :: [s]
                      , acceptData :: [s]
                      , transitionData :: TDict l s }
```

Then, encode this data into our nicer types.
Need to check if data is safe for DAs! **How?**

The Alphabet

Implementing semantics

Table of Contents

What is a non-deterministic automaton?

Implementing semantics

The power set construction

Table of Contents

What is a regular expression?

Implementing semantics

Seq and Star cases

Table of Contents

Deterministic Automata

Non-Deterministic Automata

For non-deterministic automata (NA), the transition for an input gives a *list* of next states, and we also allow empty input.

Non-Deterministic Automata

For non-deterministic automata (NA), the transition for an input gives a *list* of next states, and we also allow empty input.
For instance, the following represents a NA:

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

Non-Deterministic Automata

For non-deterministic automata (NA), the transition for an input gives a *list* of next states, and we also allow empty input.
For instance, the following represents a NA:

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

Our Haskell implementation of the type of NA:

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

Semantics of Non-Deterministic Automata

Definition

A NA accepts an input string u if there is a *possible* path that terminates at an accepting state.

Semantics of Non-Deterministic Automata

Definition

A NA accepts an input string u if there is a *possible* path that terminates at an accepting state.

Suppose the initial state is 1, and the only accepting state is 4:

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

Then this NA accepts

$$\epsilon, b, c, bc, \cdots$$

Algorithm for Running NA

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 NA

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:

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

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

Algorithm for Running NA

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:

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

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

Algorithm for Running NA

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:

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

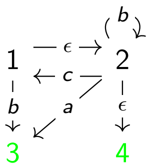
Active states: $([a], 3), ([a], 2), ([ba], 4)$.

Algorithm for Running NA

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:



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 all the states that is reachable from the current state 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

Kleene's Theorem

Theorem

The following are equivalent, for a language $l \in \mathcal{P}(X)$:

- *it is represented by a state in a finite DA*

Kleene's Theorem

Theorem

The following are equivalent, for a language $l \in \mathcal{P}(X)$:

- *it is represented by a state in a finite DA*
- *it is represented by a state in a finite NA*

Kleene's Theorem

Theorem

The following are equivalent, for a language $l \in \mathcal{P}(X)$:

- *it is represented by a state in a finite DA*
- *it is represented by a state in a finite NA*
- *it is represented by a regular expression*

Kleene's Theorem

Theorem

The following are equivalent, for a language $l \in \mathcal{P}(X)$:

- *it is represented by a state in a finite DA*
- *it is represented by a state in a finite NA*
- *it is represented by a regular expression*

in such a case, the language l is called ***regular****.*