

Haskelleene: A Very Haskell Implementation of Regular Expressions and Finite Automata Equivalence

Liam Chung, Brendan Dufty, Lingyuan Ye

31st May, 2024

Table of Contents

Introduction

Finite Automata

Kleene's Theorem

Table of Contents

Introduction

Finite Automata

Kleene's Theorem

Introduction

"Automata are pretty cool."

- Liam Chung

Motivation: What Do We Do These Things That We Do?

this is some test that is gonna go here to talk about why we do be doing things that we do be doing

Motivation: more stuff?

look at this this is more stuff that we do
wanna see a trick?
check this out:

Motivation: more stuff?

look at this this is more stuff that we do
wanna see a trick?
check this out: 😊

Table of Contents

Introduction

Finite Automata

Kleene's Theorem

Deterministic Automata

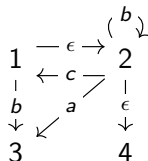
Non-Deterministic Automata

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.

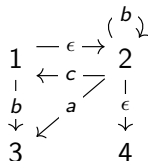
For instance, the following represents a NA:



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:



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

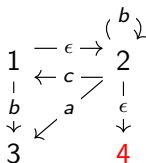
A NA accepts an input string u if there is a *possible* path that terminates at an accepting state.

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:



Then this NA accepts

$\epsilon, b, c, bc, \dots$

Algorithm for Running NA

General Idea

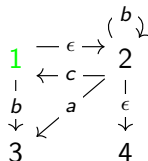
Searching paths in a finite graph in general requires a lot of computational resources. However, we do not need to output the whole path, thus we only keep track of how states transform.

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:



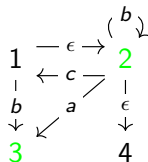
Active states: $([ba], 1)$.

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:



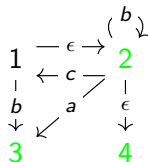
Active states: $([a], 3)$, $([ba], 2)$.

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:



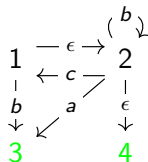
Active states: $([a], 3), ([a], 2), ([ba], 4)$.

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

Introduction

Finite Automata

Kleene's Theorem

The Theorem

Theorem

a theorem go here?