CL exercise for
Tutorial 10

Introduction

Objectives

You will complete the implementation of the determinization construction for FSMs.

Tasks

Exercise 1 is mandatory, with code answers.
Exercise 2 is optional, with written answers.

Submission

Submit a file `cl-tutorial-10` (image or pdf) with your answers that do not require programming, and the file `CLTutorial10.hs` with your Haskell code.

Deadline

16:00 Friday 2 December

NOTE unusual deadline day owing to rescheduled lectures.

Reminder

Good Scholarly Practice

Please remember the good scholarly practice requirements of the University regarding work for credit.

You can find guidance at the School page

https://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct.

This also has links to the relevant University pages. Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.

FSMs in Haskell, continued

This tutorial carries on directly from last week's tutorial. In the `CLTutorial10.hs` file we have included our solutions to last week's problems. If you prefer your own, feel free to replace them.

Make sure you have understood the subset (or powerset) construction as explained on the slides. To remind you, it is:

- The states of the DFA will be 'superstates' of the original – each superstate is a *set* of states of the original machine. We need to include each set of active states reachable from the starting set of states.
- The alphabet of symbols is unchanged.
- The DFA will have an `a`-transition from superstate `superq` to superstate `superq'` whenever `superq'` is the set of `a`-targets of states in `superq`.
- The accepting (super)states of the DFA are those which contain some accepting state of the original FSM.
- The initial (super)state is just the set of start states of the original FSM.

Figure 1 shows two FSMs, one where the states are identified by integers, `m1 :: FSM Int` , and one where the states are characters, `m2 :: FSM Char` .

For instance, converting `m1 :: FSM Int` in Figure 1 yields `dm1 :: FSM (Set Int)` in Figure 2. (Haskell shows the set $\{1, 2, 3\}$, for example, as `fromList[1,2,3]` .)

Note how we take advantage of the fact that an FSM can have states of any type: the states of the FSM are integers and the states of the corresponding DFA are sets of integers; superstates are represented explicitly. (This is exactly why we made the type of an FSM parameterized by the type of the state.)

Note also how we're saving typing by writing lists of characters as strings: `"ab"` instead of `['a','b']`.

Exercise 1 –mandatory–marked–

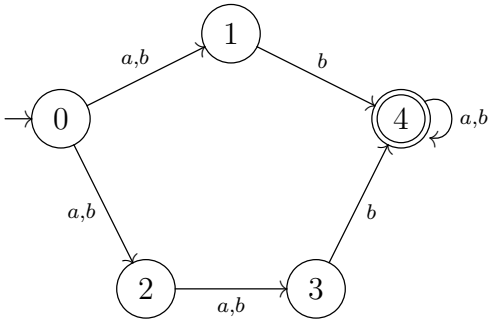
In this multipart exercise, you will complete the functions need to transform an FSM into a DFA.

1. Complete the definition of this function so that given an FSM, a source superstate, and a symbol, it returns the target superstate.

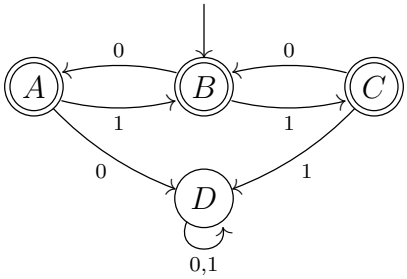
```
ddelta :: (Ord q) => FSM q -> (Set q) -> Char -> (Set q)
ddelta (FSM  qs as ts ss fs ) source  sym = undefined
```

The *target superstate* for a source superstate `source` , and a symbol, `sym` , is the set of states, `t` , such that, for some state, `s` in `source` , the machine has a `sym` -labelled transition from `s` to `t` . For example,

```
*CLTutorial10> ddelta m1 (fromList[0]) 'b'
fromList [1,2]
*CLTutorial10> ddelta m1 (fromList[1,2]) 'b'
fromList [3,4]
```

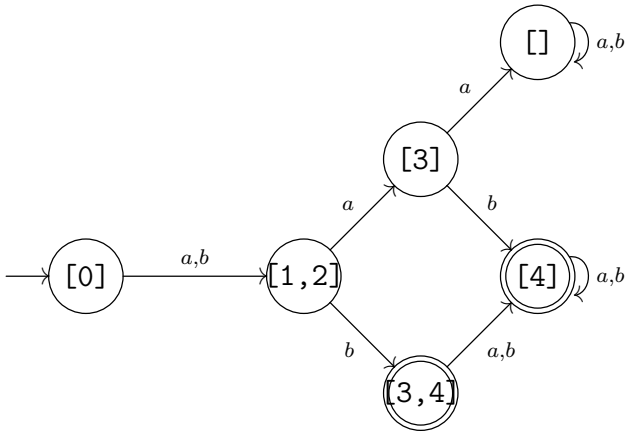



```
m1 :: FSM Int
m1 = mkFSM
     [0,1,2,3,4] -- states
     "ab"        -- symbols
     [ (0,'a',1), (0,'b',1), (0,'a',2), (0,'b',2), (1,'b',4)
     , (2,'a',3), (2,'b',3), (3,'b',4), (4,'a',4), (4,'b',4) ]
     [0]  -- starting
     [4]  -- accepting
```




```
m2 :: FSM Char
m2 = mkFSM
     "ABCD"      -- states
     "01"        -- symbols
     [('A', '0', 'D'), ('A', '1', 'B'), ('B', '0', 'A'), ('B', '1', 'C'),
      ('C', '0', 'B'), ('C', '1', 'D'), ('D', '0', 'D'), ('D', '1', 'D')]
     "B"   -- starting
     "ABC" -- accepting
```

Figure 1: Two finite state machines




```
dm1 :: FSM [Int]
dm1 = mkFSM
      [[],[0],[1,2],[3],[3,4],[4]] -- states
      "ab"                         -- symbols
      [([],   'a',[]),    ([],   'b',[])
      ,([0],  'a',[1,2]), ([0],  'b',[1,2])
      ,([1,2],'a',[3]),   ([1,2],'b',[3,4])
      ,([3],  'a',[]),    ([3],  'b',[4])
      ,([3,4],'a',[4]),   ([3,4],'b',[4])
      ,([4],  'a',[4]),   ([4],  'b',[4])]
      [[0]]       -- starting
      [[3,4],[4]] -- accepting
```

Figure 2: DFA corresponding to an FSM

```
*CLTutorial10> ddelta m1 (fromList[3,4]) 'b'
fromList [4]
```

Hint: The transition is computed by applying the `transition` function defined above.

2. If the FSM has n states, then there are 2^n possible superstates that might appear in the DFA, but we need not consider all of these. We only care about the superstates that are reachable from the start state. In the next two questions, we'll compute which states are reachable. Complete the function `next` so that, given an FSM and a list of superstates, it returns the set of superstates that can be reached in a single transition from any of these.

```
next :: (Ord q) => FSM q -> Set(Set(q)) -> Set(Set(q))
next (FSM qs as ts ss fs) supers =
  fromList [ undefined | super <- toList supers, sym <- toList as ]
```

Hint: The value can be computed by applying `ddelta` to each superstate in the set and each symbol in the alphabet.

For example, the value

```
*CLTutorial10> next m1 (fromList[fromList[0],fromList[1,2]])
fromList [fromList [1,2],fromList [3],fromList [3,4]]
```

can be computed from the following calls

```
*CLTutorial10> ddelta m1 (fromList[0]) 'a'
fromList[1,2]
*CLTutorial10> ddelta m1 (fromList[0]) 'b'
fromList[1,2]
*CLTutorial10> ddelta m1 (fromList[1,2]) 'a'
fromList[3]
*CLTutorial10> ddelta m1 (fromList[1,2]) 'b'
fromList[3,4]
```

Further hint: Use a comprehension with two generators to apply `ddelta` to each superstate in the input list of superstates, and to each symbol in the alphabet. (Look again at the definition of `cartesianProduct` given in Section 3.)

We also need a function `reachable` such that, given an FSM and a list of superstates, it returns the list of every superstate that can be reached by applying any (natural) number of transitions to some superstate in the list. (N.B. Zero is a natural number.) (Observe that we again use an @-pattern to bind `fsm` , so you can call any function that takes the machine as a parameter.) We will provide this one for you, but make sure you understand it.

For example

```
*CLTutorial10> reachable m1 (fromList[fromList[0]])
fromList [fromList [],fromList [0],fromList [1,2],fromList [3],fromList [3,4],fromList
```

Hint: The value of the call above is computed starting with the following sequence of calls to `next`.

```
*CLTutorial10> next m1 (fromList[fromList[0]])
fromList [fromList [1,2]]
*CLTutorial10> next m1 (fromList[fromList[0], fromList[1,2]])
fromList [fromList [1,2],fromList [3],fromList [3,4]]
```

```
*CLTutorial10> next m1 (fromList[fromList[0], fromList[1,2], fromList[3], fromList[3,4]
fromList [fromList [],fromList [1,2],fromList [3],fromList [3,4],fromList [4]]
```

The function is:

```
reachable :: (Ord q) => FSM q ->  Set(Set(q)) -> Set(Set(q))
reachable fsm@(FSM qs as ts ss fs) supers =
  let new = next fsm supers \\ supers
  in if null new then supers else reachable fsm (supers \/ new)
```

In general, one repeatedly applies `next` to extend the list until there is no further change. At each stage we can check whether `next` has produced any new superstates. We may use the set difference operation, `\\` and `null` to do this. If there are no new superstates we are done – otherwise add these to our list and continue.

Notice that if we start from the list containing just the initial superstate, `reachable` will return every superstate that is reachable in the equivalent DFA.

3. Complete this function so that it takes an FSM and a set `supers` of superstates and returns the subset (of `supers`) of accepting states.

```
dfinal :: (Ord q) => FSM q -> Set(Set(q)) -> Set(Set(q))
dfinal  fsm@(FSM qs as ts ss fs) supers = undefined
```

Hint: The logical functions `or and any all` that we've used with lists of things also work with sets of things; you can also use `filterS` . The functions `filterS` ,`mapS` etc. that work for sets, pretty much the `Prelude` versions of `filter` , and `map` do for lists

Remember that a superstate is final if it includes a final state of the original FSM. Write a function that given a superstate determines whether it contains a final state. Then use `S.filter` to select all final superstates from the set.

Example,

```
*CLTutorial10> dfinal m1 (fromList[fromList[],fromList[0],fromList[1,2],fromList[3],fromList[
fromList [fromList [3,4],fromList [4]]
```

4. Complete this function which should take an FSM and a list of superstates and, for each symbol in the alphabet of the FSM, return a transition for each superstate in the list.

```
dtrans :: (Ord q) => FSM q -> Set(Set q) -> Set(Trans (Set q))
dtrans fsm@(FSM qs as ts ss fs) supers =
  fromList [ (q, s, undefined) | q <- toList supers, s <- toList as ]
```

For example,

```
*CLTutorial10> dtrans m1(fromList[fromList[],fromList[0],fromList[1,2],fromList[3],fromList[3
fromList [(fromList [],'a',fromList []),(fromList [],'b',fromList[]),(fromList [0],'a',fromLi
,(fromList [0],'b',fromList [1,2]),(fromList [1,2],'a',fromList[3]),(fromList [1,2],'b',fromL
,(fromList [3],'a',fromList []),(fromList [3],'b',fromList[4]),(fromList [3,4],'a',fromList [
,(fromList [3,4],'b',fromList [4]),(fromList [4],'a',fromList [4]),(fromList [4],'b',fromList
```

Hint: The target of each transition can be computed using `ddelta` . For example,

```
*CLTutorial10> ddelta m1 (fromList[0]) 'a'
fromList [1,2]
*CLTutorial10> ddelta m1 (fromList[0]) 'b'
```

```
fromList [1,2]
*CLTutorial10> ddelta m1 (fromList[1,2]) 'a'
fromList [3]
*CLTutorial10> ddelta m1 (fromList[1,2]) 'b'
fromList [3,4]
```

Further hint. The answer template is a comprehension with two generators, one iterating over the list of superstates and one iterating over the alphabet.

All being well, this function will now take an FSM and return the corresponding DFA:

```
toDFA fsm@(FSM qs as ts ss fs) = FSM qs' as' ts' ss' fs'
  where
  qs' = reachable fsm ss'
  as' = as
  ts' = dtrans fsm qs'
  ss' = singleton ss
  fs' = dfinal fsm qs'
```

For example, `toDFA m1` returns `dm1` .

Exercise 2 –optional—marked–

In this written exercise, we shall think about concatenation a bit.

In lectures, we defined concatenation on NFAs using ϵ-transitions. Why didn't we define it on DFAs?

In the following, a bold verb indicates something you should **include** in your written answer.

Consider the one-word languages $L_1 = \{ ab \}$ and $L_2 = \{ ba \}$.

1. **Draw** DFAs M_1 and M_2 that accept these languages. Use the black hole convention to ignore unwanted transitions.

   Suppose we define the following operation $\circ$, which takes two DFAs (omitting black hole states) and gives a (perhaps in fact deterministic) NFA:
   Given DFAs $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q', \Sigma, \delta', q_0', F')$ where $Q \cap Q' = \varnothing$, let $M \circ M' = (Q \cup Q' - \{q_0'\}, \delta'', \{q_0\}, F')$, where:

$$(q, a, q') \in \delta'' \text{ iff } \begin{cases} q, q' \in Q \text{ and } \delta(q, a) = q', \text{ or} \\ q, q' \in Q' - \{q_0'\} \text{ and } \delta'(q, a) = q', \text{ or} \\ q \in F \text{ and } \delta'(q_0', a) = q', \text{ or} \\ q' \in F \text{ and } \delta'(q, a) = q_0' \end{cases}$$

   or put in words, we put M and M' side by side, replace every transition from/to q_0' by transitions from/to every accepting state of M, and remove q_0', start at M, and accept when M' does.

2. **Draw** $M_1 \circ M_2$, and **verify** that it accepts the language $L_1 L_2$.

3. The NFA $M \circ M'$ may or may not be deterministic. **State, with justification,** necessary and sufficient conditions on M and M' for $M \circ M'$ to be deterministic. **Show that** if $M \circ M'$ is deterministic, then it accepts the concatenation $L(M)L(M')$.

Challenge questions: –not marked–

There is no need to submit answers for these, but you may if you wish!

4. Can you **give an example** of DFAs M_3 and M_4 such that $L(M_3 \circ M_4) \neq L(M_3)L(M_4)$?

5. In the above, we ignored black hole states, that is, we treated the DFAs as deterministic NFAs. **Discuss** what changes if we insist that black hole states are included in the arguments of $\circ$.

6. Investigate the literature (i.e. use Google:–) and **answer** the following question: if M and M' are DFAs of size $O(n)$, what is the worst case size of the smallest DFA accepting $L(M)L(M')$?