Finite State Machines
Informatics 1 – Introduction to Computation
Functional Programming Tutorial 9

Week 9 (11 – 15 Nov.)

Assessment is formative, meaning that marks from coursework do not contribute to the final mark. But coursework is not optional. If you do not do the coursework you are unlikely to pass the exams.

Attendance at tutorials is obligatory; please send email to lambrose@ed.ac.uk if you cannot join your assigned tutorial.

1 Finite State Machines in Haskell

In the Computation & Logic part of the course, you've learned about finite state machines (FSM). You know that a deterministic finite-state automaton (DFA)is an FSM with only a single start state, and, for each state/symbol pair q, x, a unique state q' with a transition (q, x, q'). We will not use ε-transitions in this tutorial.

You will use the subset construction to transform any FSM into an equivalent DFA.

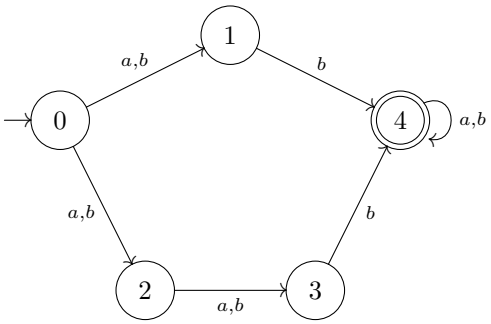
1.1 Finite State Machines over arbitrary states

Here is the type we'll use for FSMs whose states have type `q`, where `q` might be any type:

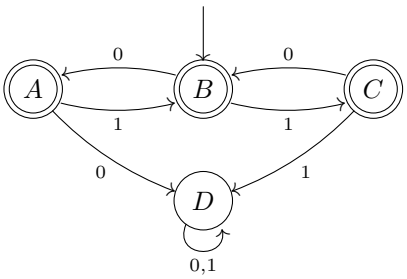
```
type FSM q = ([q], Alphabet, [q], [q], [Transition q])
type Alphabet = [Char]
type Transition q = (q, Char, q)
```

In this assignment, an FSM is a five-tuple `(k,a,s,f,t)`, consisting of: the universe of all states (`k`, a list of states), the alphabet (`a`, a list of characters), the start states (`s`, a list of states), the final states (`f`, a list of states), and the transitions (`t`, a list of transitions). Each transition `(q,x,q')` has a source state `q`, a symbol `x`, and a target state `q'`. (This is closely related to, but in a slightly different style from, the Haskell representation of FSM in CL tutorial 8.)

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`.




```
m1 :: FSM Int
m1 = ([0,1,2,3,4],
      ['a','b'],
      [0],
      [4],
      [(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)])
```




```
m2 :: FSM Char
m2 = (['A','B','C','D'],
      ['0','1'],
      ['B'],
      ['A','B','C'],
      [('A', '0', 'D'), ('A', '1', 'B'),
       ('B', '0', 'A'), ('B', '1', 'C'),
       ('C', '0', 'B'), ('C', '1', 'D'),
       ('D', '0', 'D'), ('D', '1', 'D')])
```

Figure 1: Two finite state machines

Exercise 1

Define five functions to retrieve the five components of a machine.

```
states :: FSM q -> [q]
alph   :: FSM q -> Alphabet
start  :: FSM q -> [q]
final  :: FSM q -> [q]
trans  :: FSM q -> [Transition q]
```

For example,

```
*Main> states m1
[0,1,2,3,4]
*Main> final m2
"ABC"
```

Hint: Use a pattern `(k,a,s,f,t)` as the argument of each function.

Exercise 2

Write a function that given an FSM, a set of source states, and a symbol, returns a list of all states that are the target of a transition from any of the given source states labelled with the given symbol.

```
delta :: (Eq q) => FSM q -> [q] -> Char -> [q]
```

(The type declaration has a clause `(Eq q)` because you will need to use equality (`==`) to compare states.) For example,

```
*Main> delta m1 [0] 'a'
[1,2]
*Main> delta m1 [1,2] 'a'
[3]
*Main> delta m1 [3] 'b'
[4]
*Main> delta m2 ['B'] '0'
"A"
```

Exercise 3

Write a function that given an FSM and a string returns `True` when the FSM accepts the string. The function should work with any FSM, deterministic or otherwise.

```
accepts :: (Eq q) => FSM q -> String -> Bool
```

For example,

```
*Main> accepts m1 "aaba"
True
*Main> accepts m2 "001"
False
```

Hint: Here is a skeleton of the function definition:

```
accepts :: (Eq q) => FSM q -> String -> Bool
accepts m xs  =  acceptsFrom m (start m) xs

acceptsFrom :: (Eq q) => FSM q -> q -> String -> Bool
acceptsFrom m q []      =  or [ r `elem` final m | r <- q ]
acceptsFrom m q (x:xs)  =  ...
```

The function `acceptsFrom` returns true if and only if there is a trace for the given string, from one of the given states to one of the final states. For example, machine `m1`, from its only start state, accepts the string `"aab"`.

```
*Main> acceptsFrom m1 [0] "aab"
True
```

We previously saw that

```
*Main> delta m1 [0] 'a'
[1,2]
```

Hence, from state `0`, on seeing the symbol `'a'`, the machine can move to either of state `1` or state `2`. We can recursively use `acceptsFrom` to determine if the remaining string `"ab"` is accepted in either of these states.

```
*Main> acceptsFrom m1 [1, 2] "ab"
True
```

Note that it is in fact not accepted in state 1, but is accepted in state 2.

```
*Main> acceptsFrom m1 [1] "ab"
False
*Main> acceptsFrom m1 [2] "ab"
True
```

Since the remaining string is accepted in at least one of the subsequent states, the original call succeeds.

1.2 Converting any FSM to a DFA

To convert an FSM into an equivalent DFA, we use the "powerset construction." The machine is constructed as follows:

- The states of the DFA will be "superstates" of the original—each superstate is a *set* of states of the original FSM.
- The DFA will have a transition from superstate `superq` to superstate `superq'` whenever every state in `superq'` is the target of some state in `superq`.
- The accepting (super)states of the DFA are those which include some accepting state of the original FSM.
- The initial (super)state is just the set of start states of the original FSM.

For instance, converting the first FSM in Figure 1 yields the DFA in Figure 2.

```
m1  :: FSM Int
dm1 :: FSM [Int]
```

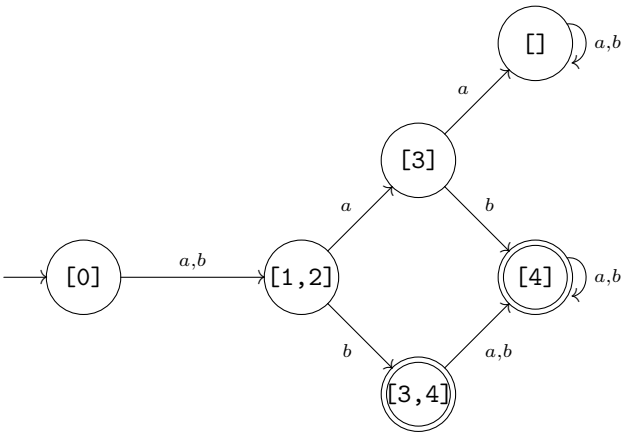
Note how we take advantage of the fact that an FSM can have states of any type: the states of the FSM are integers and the states of the corresponding DFA are lists of integers, so superstates can be represented explicitly. (This is exactly why we made the type of an FSM parameterized by the type of the state.) Our goal is to write a Haskell function that given `m1` as input will produce `dm1` as output.

Exercise 4

We use lists to represent sets. So that it is easy to compare sets, we will always represent a set by a canonical list that contains the states *in order* with *no duplicates*. Write a function that converts a list of states to its canonical form.

```
canonical :: (Ord q) => [q] -> [q]
```

(The `(Ord q) =>` clause is in the type because you will need to assume there is some order on the states.) For example,




```
dm1 :: FSM [Int]
dm1 =  ([[],[0],[1,2],[3],[3,4],[4]],
        "ab",
        [[0]],
        [[3,4],[4]],
        [([],   '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])])
```

Figure 2: DFA corresponding to an FSM

```
*Main> canonical [1,2]
[1,2]
*Main> canonical [2,1]
[1,2]
*Main> canonical [1,2,1]
[1,2]
```

Hint. Use the library functions `List.sort` and `List.nub`.

Exercise 5

Write a function that given an FSM, a source superstate, and a symbol, returns the target superstate.

```
ddelta :: (Ord q) => FSM q -> [q] -> Char -> [q]
```

The *target superstate* is the set of states to which the machine can move, starting from one of the source states, given the input symbol. For example,

```
*Main> ddelta m1 [0] 'b'
[1,2]
*Main> ddelta m1 [1,2] 'b'
[3,4]
*Main> ddelta m1 [3,4] 'b'
[4]
```

Important: The target superstate must be given in its canonical form. *Hint:* The transition is computed by applying the delta function to the given superstate and then making the result canonical.

For example, `ddelta m1 [0] 'b'` is computed from

```
*Main> delta m1 [0] 'b'
[1,2]
```

and `ddelta m1 [1,2] 'b'` is computed from

```
*Main> delta m1 [1,2] 'b'
[4,3]
```

Exercise 6

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.

Write a function `next` that, given an FSM and a list of superstates, finds all of the superstates that can be reached in a single transition from any of these and adds these reachable superstates to the input list.

```
next :: (Ord q) => FSM q -> [[q]] -> [[q]]
```

Each superstate must be canonical, and there should be no duplicates in the list.

For example,

```
*Main> next m1 [[0]]
[[0],[1,2]]
*Main> next m1 [[0],[1,2]]
[[0],[1,2],[3],[3,4]]
*Main> next m1 [[0],[1,2],[3],[3,4]]
[[],[0],[1,2],[3],[3,4],[4]]
*Main> next m1 [[],[0],[1,2],[3],[3,4],[4]]
[[],[0],[1,2],[3],[3,4],[4]]
```

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

```
*Main> next m1 [[0],[1,2]]
[[0],[1,2],[3],[3,4]]
```

can be computed from the following calls

```
*Main> ddelta m1 [0] 'a'
[1,2]
*Main> ddelta m1 [0] 'b'
[1,2]
*Main> ddelta m1 [1,2] 'a'
[3]
*Main> ddelta m1 [1,2] 'b'
[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; don't forget to add the input list of superstates to the result, and make sure that no superstate is added twice.

Exercise 7

Write a function that given an FSM and a list of superstates adds to the list any other superstates that can be reached by applying any number of transitions to any superstate in the list.

```
reachable :: (Ord q) => FSM q -> [[q]] -> [[q]]
```

For example

```
*Main> reachable m1 [[0]]
[[0],[1,2],[3],[3,4],[],[4]]
```

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

```
*Main> next m1 [[0]]
[[0],[1,2]]
*Main> next m1 [[0],[1,2]]
[[0],[1,2],[3],[3,4]]
*Main> next m1 [[0],[1,2],[3],[3,4]]
[[],[0],[1,2],[3],[3,4],[4]]
*Main> next m1 [[],[0],[1,2],[3],[3,4],[4]]
[[],[0],[1,2],[3],[3,4],[4]]
```

In general, one repeatedly applies `next` to extend the list until there is no further change.

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

Exercise 8

Write a function that takes an FSM and a list of superstates and returns a list of those that are final (accepting) in the DFA.

```
dfinal :: (Ord q) => FSM q -> [[q]] -> [[q]]
```

Remember that a superstate is final if it contains a final state of the original FSM. For example,

```
*Main> dfinal m1 [[],[0],[1,2],[3],[3,4],[4]]
[[3,4],[4]]
```

Hint: First write a function that given a superstate determines whether it contains a final state, using the `or` function and a comprehension. Then use it to select all final superstates from the list.

Exercise 9

Write a function that takes an FSM and a list of superstates and returns a transition for each superstate in the list and each symbol in the alphabet of the FSM.

```
dtrans :: (Ord q) => FSM q -> [[q]] -> [Transition [q]]
```

For example,

```
*Main> dtrans m1 [[],[0],[1,2],[3],[3,4],[4]]
[([],'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])]
```

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

```
*Main> ddelta m1 [0] 'a'
[1,2]
*Main> ddelta m1 [0] 'b'
[1,2]
*Main> ddelta m1 [1,2] 'a'
[3]
*Main> ddelta m1 [1,2] 'b'
[3,4]
```

Further hint. Use a comprehension with two generators, one iterating over the list of superstates and one iterating over the alphabet.

Exercise 10

Write a function that takes an FSM and returns the corresponding DFA.

```
deterministic :: (Ord q) => FSM q -> FSM [q]
```

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

Hint: Use `reachable` to compute the set of states, use the same alphabet as the given FSM, use as the start state the superstate containing only the start state of the FSM, use `dfinal` to compute the final states, and `dtrans` to compute the transitions.

2 Optional Material

In this section you are asked to implement a FSM which checks whether a string matches a regular expression. There are some QuickCheck properties already defined to help you along. For all of these questions, we use `['a'..'z']` as the alphabet.

Exercise 11

(a) Write a function `charFSM :: Char -> FSM Bool` that given a character `x` returns an FSM that accepts only the string `[x]`.

(b) Write a function `emptyFSM :: FSM ()` that returns an FSM that accepts no strings.

Exercise 12

The concatenation of two FSMs `A` and `B` with the same alphabet is an automaton which accepts an input word if `A` accepts some prefix of the input word and `B` accepts the rest of the word.

Implement a function

```
concatFSM :: (Ord q, Ord q') => FSM q -> FSM q' -> FSM (Either q q')
```

which returns the concatenation of the input FSMs. Your function should work on any FSMs (whether deterministic or not).

Exercise 13

(a) Write a function `intFSM :: FSM a -> FSM Int` that translates a `FSM q` (whether deterministic or not) into an equivalent `FSM Int` which has a state space `[0..n-1]`, where `n` is the number of states in the original FSM.

(b) Write a function `stringFSM :: String -> FSM Int` that returns a FSM that accepts exactly the input string.

Exercise 14

Say that a FSM is *complete* if there is at least one transition `(q,x,q')` for each state/symbol pair `q,x`.

(a) Write a function `completeFSM :: FSM a -> FSM (Maybe a)` that takes an FSM and returns an equivalent complete FSM. Hint: Use `Nothing` as a 'black hole' state, and add the missing transitions.

(b) The union of two FSMs `A` and `B` is an FSM which accepts a word if either `A` or `B` accepts it.

Write a function

```
unionFSM :: (Ord q, Ord q') => FSM q -> FSM q' -> FSM (Maybe q, Maybe q')
```

which returns the union of the input FSMs. Your function should work on both deterministic and non-deterministic FSM.

Hint: Use `completeFSM`.

A state of the union of two complete FSM consists of a pair `(q,q')` where `q` is a state of `A` and `q'` is a state of `B`.

A transition in the union FSM is a triple `((q_0,q_1),char,(q_0',q_1'))` such that `(q_0,char,q_0')` and `(q_1,char,q_1')` are transitions in `A` and `B` respectively.

Exercise 15

(a) The Kleene star closure of a FSM `A` accepts any input word which consists of a concatenation of words accepted by `A`. This includes the empty concatenation: the empty string is accepted.

Write a function `star :: FSM q -> FSM q` which returns the Kleene star closure of the input automaton. Your function should work on both any FSM (whether deterministic or not).

(b) Try out your code by writing some regular expressions of your own and testing them on sample strings.

Exercise 16

In the last part of this tutorial, you will implement two additional operations, complement and intersection.

(a) Write a function `complementFSM :: (Ord q) => FSM q -> FSM (Maybe q)` that returns an FSM which accepts an input if and only if the input FSM rejects it. Use the provided QuickCheck property to test your function.

(b) The intersection of two FSMs `A` and `B` is a FSM which accepts a word if and only if both `A` and `B` accept it.

Implement a function

```
intersectFSM :: (Ord q, Ord q') => FSM q -> FSM q' -> FSM (q,q')
```

which returns the intersection of the input FSMs. Your function should work on any FSM (whether deterministic or not).

(c) Use the QuickCheck properties at the bottom of tutorial9.hs to write your own tests.