Finite State Machines
Informatics 1 – Functional Programming: Tutorial 8

Heijltjes, Wadler

Due: The tutorial of week 10 (26/27 November)

Please attempt the entire worksheet in advance of the tutorial, and bring with you all work, including (if a computer is involved) printouts of code and test results. Tutorials cannot function properly unless you do the work in advance.

You may work with others, but you must understand the work; you can't phone a friend during the exam.

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 let your tutor know if you cannot attend.

Finite State Machines in Haskell

In the Computation & Logic part of the course, you've learned about finite state machines (FSMs), both deterministic (D-FSMs) and nondeterministic (N-FSMs), and you've learned how the latter can be transformed into the former.

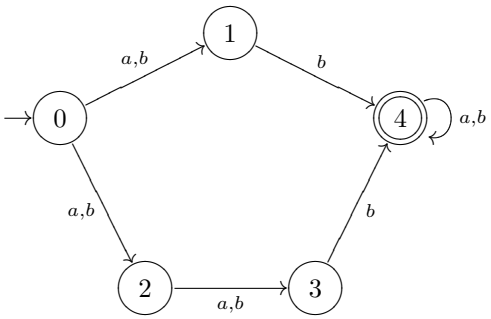
Finite State Machines over arbitrary states

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

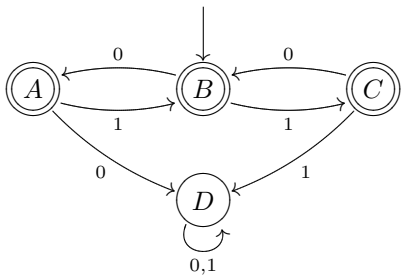
```
type FSM q = ([q], Alphabet, q, [q], [Transition q])
type Alphabet = [Char]
type Transition q = (q, Char, q)
```

In this assignment, a FSM is a five-tuple `(u,a,s,f,t)`, consisting of: the universe of all states (`u`, a list of states), the alphabet (`a`, a list of characters), the start state (`s`, a state), 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'`. (We use `u` rather than `q` for the universe of all states, since we use `q` for individual states.)

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

Exercises

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 `(u,a,s,f,t)` as the argument of each function.

2. Write a function that given an FSM, a source state, and a symbol, returns a list of all states that are the target of a transition for the given source state and 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 m2 'B' '0'
"A"
```

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 []      =  q `elem` final m
acceptsFrom m q (x:xs)  =  ...
```

   The function `acceptsFrom` returns true if and only if it accepts the given string starting in the given state. For example, machine `m1` in its 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 "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.

Further hint: To fill in the `...`, use a list comprehension that iterates over the states returned by `delta` and uses `acceptsFrom` recursively to compute a list of boolean, then use an appropriate function to combine the booleans.

Converting an N-FSM to a D-FSM

To convert an N-FSM into a D-FSM, we can use an technique called the "superset" construction. The machine is constructed as follows:

- The states of the D-FSM will be "superstates" of the original—each superstate is a *set* of states of the original machine.
- The D-FSM will have a transition from superstate `superq` to superstate `superq'` whenever each state in `superq'` is the target of some state in `superq`.
- The accepting (super)states of the D-FSM are those which contain some accepting state of the original N-FSM.
- The initial (super)state is just the singleton set containing only the initial state of the original N-FSM.

For instance, converting the first N-FSM in Figure 1 yields the D-FSM 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 N-FSM are integers and the states of the corresponding D-FSM 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.

Exercises

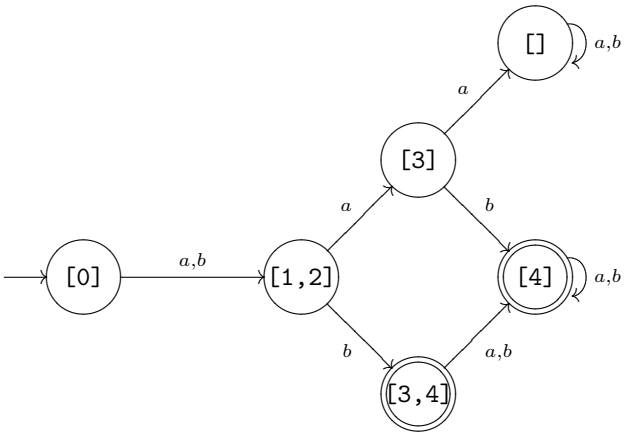
4. We use lists to represent sets. So that it is easy to compare sets, we will always represent sets by a canonical list that contains the states *in order* with *no duplicates*. Write a function that converts a list of states to its canonical form.

   ```
   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,

   ```
   *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`.




```
dm1 :: FSM [Int]
dm1 =  ([[],[0],[1,2],[3],[3,4],[4]],
        ['a','b'],
        [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: D-FSM corresponding to an N-FSM

5. Write a function that given an N-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 should be given in its canonical form.

Hint: The transition is computed by applying the delta function to each state in the given superstate and then combining and canonicalizing the results. 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 'b'
[4]
*Main> delta m1 2 'b'
[3]
```

Further hint: Use a list comprehension and possibly the library function `concat`.

The set of superstates that we include in the D-FSM need not involve every set of states from the N-FSM (Think: how many of these are there?). We only care about the sets that are reachable from the start state. In the next two questions, we'll compute which states are reachable.

Exercises

6. Write a function `next` that, given an N-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.

7. Write a function that given an N-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 D-FSM.

8. Write a function that takes a N-FSM and a list of superstates and returns a list of those that are final (accepting) in the D-FSM.

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

   Remember that a superstate is final if it contains a final state of the original N-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.

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

10. Write a function that takes an N-FSM and returns the corresponding D-FSM.

    ```
    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 N-FSM, use as the start state the superstate containing only the start state of the N-FSM, use `dfinal` to compute the final states, and `dtrans` to compute the transitions.

11. Write a QuickCheck property

    ```
    prop_deterministic :: (Ord q) => FSM q -> [Int] -> Bool
    ```

    to verify that, given a FSM `m`, `deterministic m` and `m` itself accept the same strings. You should let QuickCheck generate lists of integers and use the predefined function `makeInput` to transform these to strings over the alphabet of `m`. Use your property with FSMs `m1` and `m2` to test your definition of `deterministic`.

Optional material

A successful trace through a non-deterministic machine lists the transitions in order, alternating states and symbols. For instance, for machine `m1` there are two successful traces accepting the string `"abb"`.

```
[0,'a',1,'b',4,'b',4]
[0,'a',2,'b',3,'b',4]
```

We represent a trace as a list of steps, where each step is either a state or a symbol.

```
data Step q = State q | Symbol Char
type Trace q = [Step q]
```

Hence, the two traces above are represented as follows.

```
[State 0,Symbol 'a',State 1,Symbol 'b',State 4,Symbol 'b',State 4]
[State 0,Symbol 'a',State 2,Symbol 'b',State 3,Symbol 'b',State 4]
```

Normally a trace would print as above, but by defining a show function for steps we can suppress the constructors `State` and `Symbol`.

```
instance (Show q) => Show (Step q) where
  show (State q)   =  show q
  show (Symbol x)  =  [x]
```

The clause `(Show q) => Show (Step q)` says that we can define how to show a value of type `(Step q)` given that one already has a show function for type `q`. With these definitions, a value of trace type prints as shown.

```
*Main> [State 0,Symbol 'a',State 1,Symbol 'b',State 4,Symbol 'b',State 4]
[0,a,1,b,4,b,4]
```

Exercises

12. Write a function that given an FSM and a string returns all accepting traces for the string.

    ```
    traces :: (Eq q) => FSM q -> String -> [Trace q]
    ```

    For example,

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

    Hint. Here is a skeleton of the function definition.

    ```
    traces :: (Eq q) => FSM q -> String -> [Trace q]
    traces m xs  =  trac m (start m) xs

    trac :: (Eq q) => FSM q -> q -> String -> [Trace q]
    trac m q []      =  [ [State q] | q `elem` final m ]
    trac m q (x:xs)  =  ...
    ```

    The function `trac` returns traces for the given string starting in the given state. For example, machine `m1` in its start state has two traces for the string `"aaba"`.

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

    We previously saw that

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

Hence, from state `0` on seeing the symbol `'a'` the machine might transition to either state `1` or state `2`. We can recursively use `trac` to find traces for remaining string `"bb"` in either of these states.

```
*Main> trac m1 1 "bb"
[[1,'b',4,'b',4]]
*Main> trac m1 2 "bb"
[[2,'b',3,'b',4]]
```

The two traces returned by the original call are built by appending the start state `0` and the first symbol `'a'` to the beginning of these traces.