CSci 658: Software Language Engineering
Mealy Machine Simulator Exercise

H. Conrad Cunningham

17 February 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Advisory: The HTML version of this document may require use of a browser that supports the display of MathML. A good choice as of February 2018 is a recent version of Firefox from Mozilla.

TODO:

- Are the operations and their specifications reasonable for Haskell?
- Is an `Either` the appropriate return in the various cases?
- Should `getTransitionsFrom` return an `Either` for an invalid state?

Mealy Machine Simulator

Mealy Machine Definition

In this exercise, you are asked to design and implement Haskell modules to represent Mealy Machines and to simulate their execution.

This kind of machine is a useful abstraction for simple controllers that listen for input events and respond by generating output events. For example in an automobile application, the input might be an event such as “fuel level low” and the output might be command to “display low-fuel warning message”.

In the theory of computation, a *Mealy Machine* is a finite-state automaton whose output values are determined both by its current state and the current input. It is a *deterministic finite state transducer* such that, for each state and input, at most one transition is possible.

Appendix A of the textbook *Formal Languages and Automata*, 6th Ed., by Peter Linz (Jones & Bartlett, 2017) defines a Mealy Machine mathematically by a tuple

$$M = (Q, \Sigma, \Gamma, \delta, \theta, q_0)$$

where

Q is a finite set of internal states
Σ is the input alphabet (a finite set of values)
Γ is the output alphabet (a finite set of values)
$\delta : Q \times \Sigma \longrightarrow Q$ is the transition function
$\theta : Q \times \Sigma \longrightarrow \Gamma$ is the output function
q_0 is the initial state of M (an element of Q)

In an alternative formulation, the transition and output functions can be combined into a single function:

$$\delta : Q \times \Sigma \longrightarrow Q \times \Gamma$$

We often find it useful to picture a finite state machine as a *transition graph* where the states are mapped to vertices and the transition function represented by directed edges between vertices labelled with the input and output symbols.

Exercises

1. Design and implement a general representation for a Mealy Machine as a Haskell module implementing an abstract data type. It should hide the representation of the machine and should have, at least, the following public operations.

    - `newMachine s` creates a new machine with initial (and current) state `s` and no transitions.

      Note: This assumes that the state, input, and output sets are exactly those added with the mutator operations below. An alternative would be to change this function to take the allowed state, input, and output sets.

- `addState m s` adds a new state `s` to machine `m` and returns an `Either` wrapping the modified machine or an error message.
- `addTransition m s1 in out s2` adds a new transition to machine `m` and returns an `Either` wrapping the modified machine or an error message. From state `s1` with input `in` the modified machine outputs `out` and transitions to state `s2`.
- `addResets m` adds all reset transitions to machine `m` and returns the modified machine. From state `s1` on input `in` the modified machine outputs `out` and transitions to state `s2`. This operation makes the transition function a total function by adding any missing transitions from a state back to the initial state.
- `setCurrent m s` sets the current state of machine `m` to `s` and returns an `Either` wrapping the modified machine or an error message.
- `getCurrent m` returns the current state of machine `m`.
- `getStates m` returns a list of the elements of the state set of machine `m`.
- `getInputs m` returns a list of the input set of machine `m`.
- `getOutputs m` returns a list of the output set of machine `m`.
- `getTransitions m` returns a list of the transition set of machine `m`. Tuple `(s1,in,out,s2)` occurs in the returned list if and only if, from state `s1` with input `in`, the machine outputs `out` and moves to state `s2`.
- `getTransitionsFrom m s` returns an `Either` wrapping a list of the set of transitions enabled from state `s` of machine `m`or an error message.

2. Given the above implementation for a Mealy Machine, design and implement a separate Haskell module that simulates the execution of a Mealy Machine. It should have, at least, the following new public operations.

   - `move m in` moves machine `m` from the current state given input `in` and returns an `Either` wrapping a tuple `(m',out)` or an error message. The tuple gives the modified machine `m'` and the output `out`.
   - `simulate m ins` simulates execution of machine `m` from its current state through a sequence of moves for the inputs in list `ins` and returns an `Either` wrapping a tuple `(m',outs)` or an error message. The tuple gives the modified machine `m'` after the sequence of moves and the output list `outs`.

3. Implement a Haskell module that uses a different representation for the Mealy Machine. Make sure the simulator module still works correctly.

Acknowledgements

In Spring 2017, I adapted this document from an assignment given in the Scala-based offering of CSci 555 Functional Programming in Spring 2016. I edited the format slightly in Spring 2018.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

Concepts

TODO