Lecture 2

Inf1A: Deterministic Finite State Machines

2.1 Introduction

In this lecture we will focus a little more on *deterministic* Finite State Machines. Also, we will be mostly concerned with Finite State Machines which are *acceptors* (ie, have no output alphabet) for this lecture. The idea is that we will explore deterministic FSMs in more depth, giving formal definitions of things that were informally discussed in Lecture 1.

We will also discuss the *binary representation* of natural numbers in this lecture. All modern digital computers store data such as natural numbers, floating point numbers, and letters of the alphabet in binary representation: that is, in terms of sequences of `0`s and `1`s.

2.2 Formal Definitions

Before we launch into our formal definition of deterministic FSMs, it is useful to know a bit about some of the words we will use throughout the definition.

An alphabet Σ is just some set containing a finite number of distinct symbols. When we talk about an alphabet, we are usually interested in *strings* over that alphabet. A *string* over Σ is simply a sequence of *any number* of symbols taken from Σ. We write Σ^* to represent the set of *all strings* over Σ. For example, if the alphabet is $\Sigma = \{\mathtt{c}, \mathtt{d}\}$, then all the following are elements of Σ^*:

$$\epsilon, \mathtt{c}, \mathtt{d}, \mathtt{cc}, \mathtt{cd}, \mathtt{dc}, \mathtt{dd}, \mathtt{ccc}, \mathtt{ccd}, \mathtt{cdc}, \ldots$$

In general Σ^* contains strings of *any* length which are made up of `c`s and `d`s. Notice that one of the strings is special. This is the ϵ string, which is the *empty string*, which contains no symbols from Σ. The empty string ϵ is an important member of Σ^* for any alphabet Σ.

For any alphabet Σ, a *language* over Σ^* is a subset L of Σ^* (a set L such that every string in L belongs also to Σ^*). For example, the set $L_{\mathtt{c}}$ containing all strings made up of `c`s is a language over $\{\mathtt{c}, \mathtt{d}\}$. The set $L' = \{\mathtt{c}, \mathtt{ddc}\}$ is also a language over $\{\mathtt{c}, \mathtt{d}\}$.

Here is the formal definition of an *acceptor*-type *deterministic Finite State Machine.*

Definition (Deterministic FSM)
A deterministic Finite State Machine *(D-FSM) of the* acceptor *type is a machine M defined as a 5-tuple*

$$M = (Q, \Sigma, s_0, F, \delta)$$

consisting of

1. *a finite set Q of* states
2. *a finite alphabet Σ*
3. *a distinguished* start state $s_0 \in Q$
4. *a set $F \subseteq Q$ of* accepting states, *and*
5. *a* description *of all the possible* transitions *of the FSM.*

   The usual formal definition given for deterministic FSMs states that the set of transitions of the machine should be a function $\delta : Q \times \Sigma \to Q$. *We refer to this function δ as the* transition function *of the machine M.*

Notice that item 5. of the definition given above implies that at any state q of a D-FSM, we must have exactly one transition leaving q for every different symbol $a \in \Sigma$ (because when we talk about a *function* with the domain $Q \times \Sigma$, this means that $\delta(q, a)$ must be defined for all possible values of $q \in Q$ and $a \in \Sigma$).

In practice we can be a little bit more flexible than that. Remember from Lecture 1 that our goal in defining D-FSMs was to ensure that for *every* input string $x \in \Sigma^*$, there is *at most one* trace in M for x. We achieved this with a strict definition of the transitions allowed. If $x = \epsilon$, then the fact that there are no ϵ-transitions in a D-FSM implies that there is exactly one trace for x (which is the trace s_0). If $x \neq \epsilon$ (it has at least one symbol), there is *exactly one* list of states $s_0, \ldots, s_k$ (where s_0 is the start state) such that

$$s_0, x_1, \ldots, x_k, s_k$$

is a trace of M. This is because since δ is a function, therefore for every state $q \in Q$ and every symbol a, there is *exactly one* transition leaving q labelled with a (the state q' given by $q' = \delta(q, a)$), and no ϵ-transition leaving q. Therefore at any state of the machine reading any symbol, there is no "choice" (no non-determinism) for the next state - it is given by δ. The trace above is given by $s_1 = \delta(s_0, x_1), \ldots, s_k = \delta(s_{k-1}, x_{k-1})$.

However, our transitions do not have to define a function on $Q \times \Sigma$ in order to avoid multiple traces - it is enough that for every $q \in Q$ and every $a \in \Sigma$, there is *at most one* q' such that there is an transition from q to q' while reading the symbol a. Therefore we will sometimes allow the transition function to be undefined for some elements of $Q \times \Sigma$.

Definition (Language accepted by a D-FSM)

Let $M = (Q, \Sigma, s_0, F, \delta)$ be a deterministic FSM of the acceptor. *Then the* language $L(M)$ *accepted by M is the set of strings $x \in \Sigma^*$ such that*

(i) there is a trace for x in M;

(ii) the trace ends in an accepting state (a state from F).

Observe that for any D-FSM whose set of transitions is a *function*, it is always the case that (i) above holds (there is a trace for every string $x \in \Sigma^*$). Therefore for any $x \in \Sigma^*$, x is rejected if and only if the last state in its trace is a non-accepting state (the last state is not in F).

It would be nice if every D-FSM was a equivalent to a D-FSM which satisfied the strict conditions of the Definition given above. We now show that this is indeed the case.

Let M be a deterministic FSM whose set of transitions is *not* strictly a function. Therefore there may be some strings $x \in \Sigma^*$ which are rejected (not accepted) by the machine because there is no trace for the string - this means that at some point during the computation of M on x, there is no transition leaving the current state which is labelled with the "next" symbol of the input string. Now suppose we alter M by adding a *new* state to Q - we will call this a *sink state*. The sink state will *not* be an accepting state. Now for every state q and every symbol a such that there is no transition leaving q labelled with a, we add such a transition, using the sink state as the target state. Therefore, we will direct all strings which have no trace in M to this new (non-accepting) sink state. We also add a transition from the sink state to itself labelled with all of the symbols of the alphabet Σ. This ensures that the transition function is complete on the extended state set, and that once a string is sent to the sink state, it will stay there. Therefore every string that had no trace in the machine M now has a trace in the extended FSM, and most importantly, every such string is rejected by the new machine. So the *language* of the new machine is identical to $L(M)$.

2.3 Binary numbers and Unary numbers

Throughout your years in high school you are probably used to writing the natural numbers as a sequence of *digits* from $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. These digits are known as the *decimal digits*. In the terminology of Section 2.2 we could say that the set of *decimal numbers* is the set of *strings* over the alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ (as long as we identify the empty string ϵ with the number 0). This is the language Σ^*. In fact (better), we could say that the set of *decimal numbers* is the set of *strings* over the alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ whose leading digit is not 0.

In this section we will present two alternative systems for representing natural numbers: (i) the *binary numbers* (which is the representation that digital computers use for natural numbers), and (ii) a more unusual (and *much* less compact) number system, which is the *unary number* system.

2.3.1 Decimal numbers

For any natural number[1] $n \in \mathbb{N}$, we can represent n in the decimal system by some sequence of decimal digits $d_{k-1} \ldots d_1 d_0$, where the digits $d_0, d_1, \ldots, d_{k-1} \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ are chosen so that

$$n = \sum_{i=0}^{k-1} 10^i d_i = d_0 + 10d_1 + 10^2 d_2 + \ldots + 10^{k-1} d_{k-1}.$$

[1]The natural numbers are just the counting numbers, that is: $0, 1, 2, 3, 4, \ldots$ and so on.

The value k is just a variable to represent the length of the string. This representation is the way you are used to seeing numbers written. Of course the representation is *unique* (there is exactly one decimal representation for each natural number, unless you decide to add some useless 0s after the final digit d_{k-1}). We represent 0 by the empty string ϵ (or by 0).

As we know well, the leading digits are the digits which contribute most to the value of n - we say that the digit d_{k-1} (assuming it is non-zero) is the *most-significant digit* and that d_0 is the *least-significant digit.*

2.3.2 Binary numbers

Suppose that we only are allowed to work with the binary alphabet $\{0, 1\}$, and that we need a way of representing natural numbers using strings over this alphabet (that is, we have $\Sigma = \{0, 1\}$, and we have to work with strings from Σ^*).

We show that we can represent all natural numbers by binary strings. For every binary string $b_{k-1} \dots b_1 b_0$, where $b_0, b_1, \dots, b_{k-1} \in \{0, 1\}$ (and where k is just a variable to allow us to talk about the length of the string), the number represented by $b_{k-1} \dots b_1 b_0$ is the natural number

$$n \; = \; \sum_{i=0}^{k-1} 2^i b_i \; = \; b_0 + 2b_1 + \dots + 2^{k-1} b_{k-1}.$$

For every natural number n, the *binary representation* of that number is unique, if we insist that there are no leading 0s. Again b_{k-1} is the *most significant digit* (assuming it is non-zero) and b_0 is the *least significant digit.*

It is not difficult to come up with the binary representation of a natural number n - you can find the binary number by successively dividing n by 2 and taking the remainders (each is either 0 or 1) in reverse order as the basis of the binary representation. For example, if we consider $n = 53$, then performing this procedure gives:

53		original number
26	1	dividing 53 by 2 gives div 26, rem 1
13	0	dividing 26 by 2 gives div 13, rem 0
6	1	dividing 13 by 2 gives div 6, rem 1
3	0	dividing 6 by 2 gives div 3, rem 0
1	1	dividing 3 by 2 gives div 1, rem 1
0	1	dividing 1 by 2 gives div 0, rem 1

Then, reading the remainders off in reverse, the binary representation of 53 is `110101`. It is an interesting exercise to try to give a formal argument that this procedure always constructs the correct binary representation of n.

Binary numbers play a central role in Computer Science, and it is fun to see how to add, subtract, multiply and divide binary numbers (without converting into a decimal representation). There is some information about this at the `Wikipedia` link at the end of these notes.

2.3.3 Unary numbers

Finally, we mention a very inefficient representation of natural numbers. Suppose our alphabet only contains one symbol, that is, $\Sigma = \{1\}$. Suppose we want to represent the natural numbers as strings over Σ^*. In this case, we have the following representation:

$$n = \begin{cases} \epsilon & \text{if } n = 0 \\ 1 \ldots 1 & \text{if } n \geq 1 \end{cases},$$

where in the $n \geq 1$ case, we repeat 1 exactly n times.

2.4 Deterministic FSMs to recognise odd numbers

We will now consider the task of designing a deterministic FSM to recognise odd natural numbers. We will construct three different FSMs - one for numbers in *unary notation*, one for numbers in *binary notation* and finally, an FSM for numbers in *decimal notation*.

Odd numbers in unary

To construct the machine for odd unary numbers, remember that our input alphabet is $\Sigma = \{\texttt{1}\}$. Consider what happens for the empty string ϵ - this represents 0, which is *not* odd. So the start state of our machine, labelled 1 below, will not be an accepting state.

Notice that whenever we read an extra `1` from the input string, we switch back and forth between having seen an odd number of `1`s and an even number of `1`s. Therefore we really only need to have two states for our machine. The second state, labelled 2, will represent the state of having seen an odd number of `1`s so far.

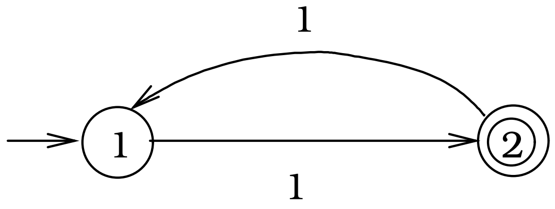

Figure 2.1: Machine to recognise odd numbers in *unary*

It is a simple task, but worthwhile, to test the machine above on some short strings over `1`*. Notice that the FSM in Figure 2.1 is deterministic, and has a complete transition function.

Odd numbers in binary

In this case our input alphabet is $\Sigma = \{\texttt{0}, \texttt{1}\}$. In order to recognise odd numbers in binary, we need to make a decision about the order in which the binary number is input to the FSM. We will require that the binary string is presented in the order of *least-significant* bit first. Now think about the binary representation described in Section 2.3. The natural number represented by the binary string $b_{k-1} \ldots b_0$ is $n = b_0 + 2b_1 + \ldots + 2^{k-1}b_{k-1}$. Notice that every term in this summation is multiplied by 2 (and greater powers of 2) except for the least significant bit b_0. Therefore the natural number n is odd *if and only if* the least significant bit is equal to `1` - if we see that the least significant bit is `1`, then we do not need to check anything else.

Here is our first attempt at doing the FSM. Notice that the start state is not an accepting state. We have a second state, labelled 2, which is accepting. We draw the transitions in such a way so that we only transition to this accepting state if the first bit read from the input string (the least significant bit) is `1`.

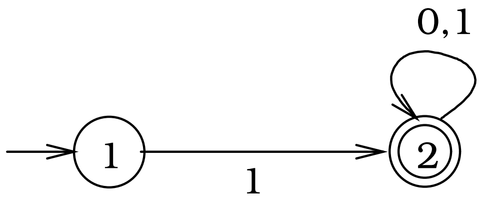

Figure 2.2: Machine to recognise odd numbers in *binary*

It is a good exercise to test the strings `101001` (the number 41) and `1000` (the number 8) on the machine in Figure 2.2. The testing phase is always useful for *debugging* FSMs.

Notice that the machine in Figure 2.2 is deterministic. There are no ϵ-transitions, and all transitions leaving the same state are labelled with different symbols (or bits, in this case).

Suppose we draw a $Q \times \Sigma$ table to represent the transitions of the FSM in Figure 2.2:

	`0`	`1`
state 1	-	state 2
state 2	state 2	state 2

Notice from this table that the set of transitions for our D-FSM of Figure 2.2 is not complete. It is not a *function* on $Q \times \Sigma$. However, using the sink state construction from the end of Section 2.2, we can construct an equivalent D-FSM with a transition function to recognise odd numbers in binary. This new D-FSM is shown in Figure 2.3. The sink state is the state 3.

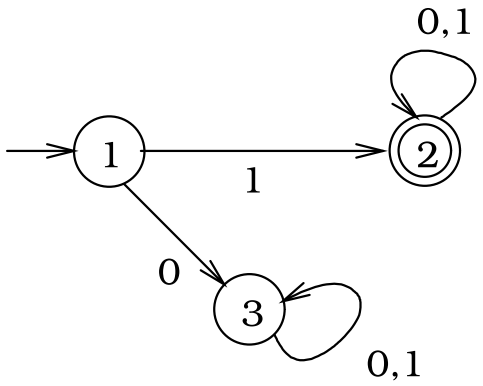

Figure 2.3: Machine 2 to recognise odd numbers in *binary*

The machine in Figure 2.3 accepts exactly the same language as the machine of Figure 2.2, which is the language of binary strings with a `1` in the least-significant position.

2.4.1 Odd numbers in decimal

Here is a machine to recognise odd numbers in decimal. It is a good idea to test it out, and to draw the transition-table.

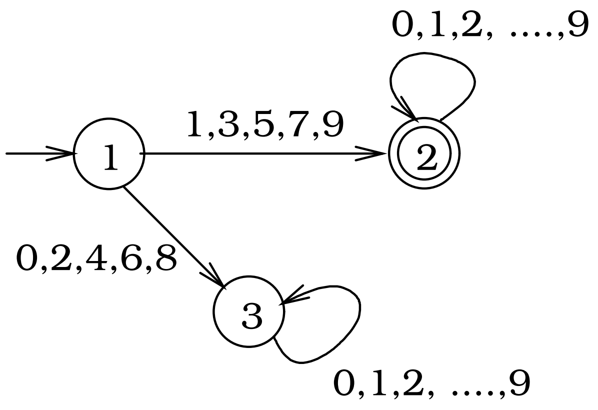

Figure 2.4: Machine to recognise odd numbers in *decimal*

2.5 Finite State Machines in Haskell

We will now see how we can implement some of our little Finite State Machines from Section 2.4 in Haskell. The first thing to say is that in general the problems that we have been modelling with Finite State Machines are much simpler than the computational tasks which you have been solving in the FP part of this course. However, one of the ways of implementing Finite State Machines involves using *mutual recursion*, which is an interesting concept in Functional programming.

Consider our first FSM for recognising odd numbers in *unary notation*. This machine has two states - the start state, named 1, indicates that the part of the string read so far consists of an *even* number of `1`s, and the other state, named 2, indicates that the part of the string read so far consists of an *odd* number of `1`s. We define two mutually recursive functions "`oddUn`" and "`evenUn`" to represent our two states.

Typing in `oddUn` with a string of `1`s returns True if the string consists of an odd number of `1`s and False if the string consists of an even number of `1`s.

It is a nice exercise to define an appropriate data type to ensure that the only lists passed to `oddUn` are lists of 1s.

```
oddUn :: [a] -> Bool
evenUn :: [a] -> Bool

oddUn [] = False
oddUn (x:xs) = evenUn xs

evenUn [] = True
evenUn (x:xs) = oddUn xs
```

2.6 Summary

In this lecture note we have

- Given a formal definition of a deterministic FSM and of the language accepted by a deterministic FSM.
- Given an introduction to *binary representation* for natural numbers as well as *unary notation*.
- Given deterministic Finite State Machines to recognise odd natural numbers represented in (i) unary notation; (ii) binary notation; (iii) and decimal notation.
- Discussed the issue of whether the set of transitions of a D-FSM form a *complete* transition function on $Q \times \Sigma$. Shown that we can ensure that a deterministic machine has its transitions described by a *transition function* by simply adding one sink state to the machine.
- Discussed the issue of *testing* FSMs for correctness.

2.7 Extra work

We didn't cover binary numbers in much depth (just enough to suit our purposes for the examples of Finite State Machines given above). Wikipedia has a very useful page on the Binary number system. It is located at:

`http://en.wikipedia.org/wiki/Binary_numeral_system`

Draw the transition table for the D-FSM for accepting odd numbers in decimal notation.

Also write a set of three mutually recursive Haskell functions to implement the machine of Figure 2.3, and another set of Haskell functions to represent the machine of Figure 2.4.

These notes have been adapted from two sets of earlier notes, the first due to Stuart Anderson, Murray Cole and Paul Jackson, and the second due to Martin Grohe and Don Sannella.

Mary Cryan, November 2004