50.002 COMPUTATIONAL STRUCTURES

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Programmability

Natalie Agus (Fall 2018)

1 Implementing FSM as ROM

We can "hardcode" FSM (called programmable machines). Given i input bits, s state bits, and o output bits we have a total combination of 2^{i+s} **input-state combo (called words), and each word has o bits as an output**.
See diagram below:

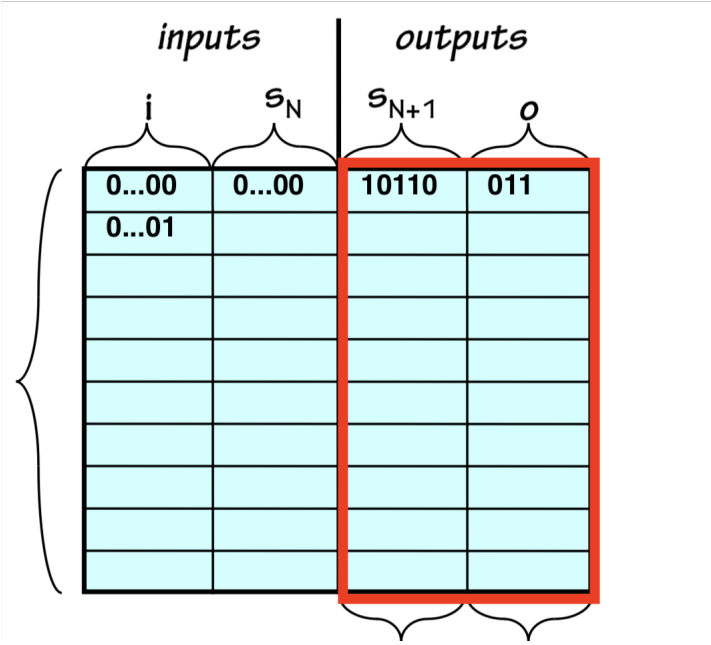


Figure 1

The number of possible FSMs that can be captured with i input bits, o output bits, and s state bits is $2^{(o+s)2^{i+s}}$ FSMs (some FSMs in these many possible FSMs may be equivalent). Explanation:

1. We have 2^{i+s} words with i input bits and s states (input-state combo)

2. Each word requires o output bits and s end-state bits
3. So, we have to fill in $(o+s)2^{i+s}$ bits in the truth table in total (the one in red box in Figure 1)
4. But, each bit can take up 2 values: 0 or 1
5. Therefore we have $2^{(o+s)2^{i+s}}$ different FSM combination

For example, if $i = 1, o = 1, s = 1$, then we can have 256 different FSMs, enumerating it and filling it to the red box we have:

1. A red box in figure 1 of 4 rows $(2^{(1+1)})$ and 2 columns $(1+1)$, thats 8 cells to fill = 8 bits
2. We can fill each of the 8 bits with either 1s or 0s
3. So we can enumerate from 00000000 to 11111111
4. That's $2^8 = 256$ possible combinations, a.k.a 256 different FSMs
5. We can just simply name them arbitrarily as FSM_1 all the way to FSM_{256}

2 FSM Limitations

Some problems cannot be computed using FSMs, so **FSMs alone is not enough to be an ultimate computing device**. A classic example that cannot be computed using FSM is *parenthesis checker*.

Reason: it requires **arbitrarily** many states because you can have as many left parenthesis as you want and you need to keep count of all left parenthesis to check if it is balanced with the right parenthesis. A FSM needs a FINITE amount of memory (states).

3 Turing Machines

The Turing model solves the FINITE problem of the FSM, so basically it solves the run-out-of-memory problem in FSMs. See diagram below of arbitrary FSM implemented as Turing machine:
Explanation:

1. The idea is to imagine you have a physical machine (like a typewriter machine or something), called the Turing Machine, that can take input of an infinitely long "tape" (its basically an array) that you can write at ('0' or '1' , or anything), and it has a **functional specification table** that describes its behavior
2. And then you have the pointer in the machine (black arrow) that represent your current state location

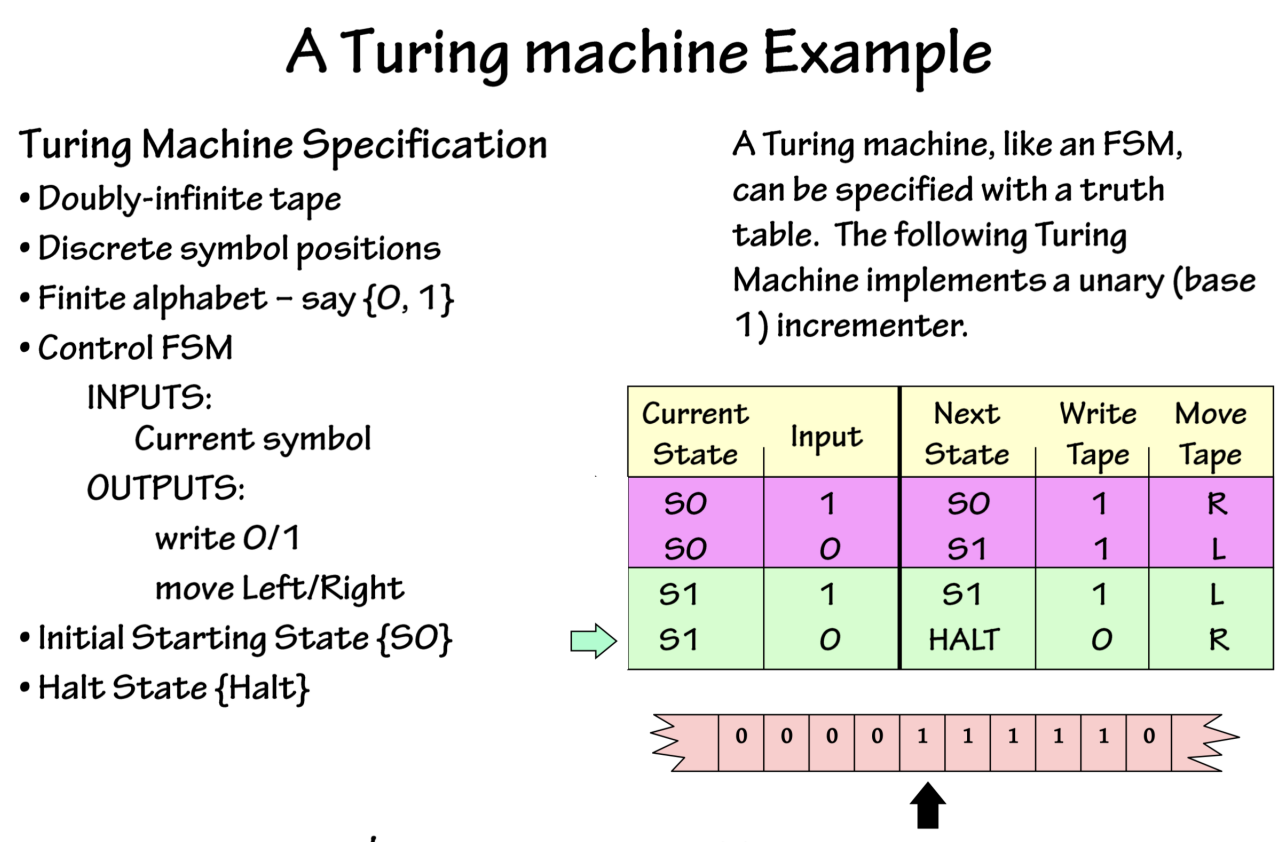


Figure 2

3. You can move the tape left and right to shift the arrow
4. Theres a HALT state where you reach the end-state
5. So at Figure 2, assume that the start state is $S0$, and as shown the black arrow is at 1, so its equivalent as saying that the input is 1
6. Then look at the functional specification table, and decide where to go next.
7. From the table, if current state is $S0$ and input is 1, you **write 1** and then move the arrow to the right
8. *It is important to write first, and then move the arrow*, not the other way around
9. Repeat until you meet a HALT as next state

4 Example: Turing Machine Tape as Integer

In this section we are going to go through a an example of how turing machine can be used as a machine that ”reads some integer”. This is just one of the many many functionalities that a Turing Machine can do. See the diagram below. The arrow represents the beginning of an arbitrary FSM called FSM_{347} (the name doesnt mean

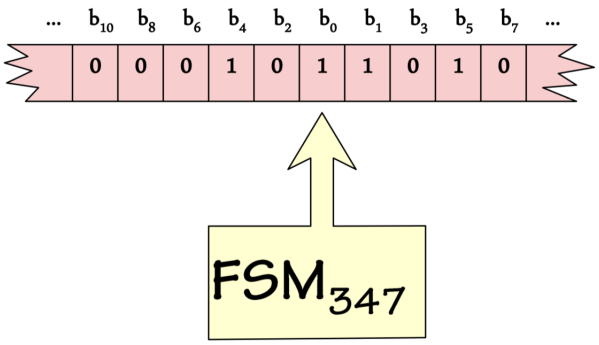


Figure 3

anything, its just some naming) that operates on a Tape number 51. To read that integer 51 on the tape (so that you know which tape you're at), you can do:

1. Take the number at the arrow head as the least significant bit, in this case thats 1.
2. Then you can move on either to the left or right depending on the number that you encode. Lets say in this case you move to the right to b_1 position and append it as the second least significant bit, so thats 11
3. After this you should move to the left to reach b_2 (because previously you chose to move to the right), and append as the third least significant bit: 011
4. Now you need to move to b_3, and repeat to append as the fourth least significant bit: 0011
5. Oscillate back to b_4, 10011
6. Oscillate to the right to b_5, 110011
7. Continuing until b_8 we have 000110011 and so on
8. So we can end up with finite integers with potentially infinite leading trail of zeroes to represent our integer
9. In this case, 000110011 is 51

5 Another example: Turing Machines as Integer Function

Given Turing Machine i (FSM_i) operating on Tape x,

1. We can have $y = T_i[x]$, where y is the output of a series of binary numbers on the the corresponding Tape x.

2. $y = T_i[x]$ be seen as a function that takes in integer x as input [1], and

3. T_i is the Turing Machine functional specs table similar to the one shown in Figure 2 that dictates the machine to go left or right according to the state.

4. y is the output after the arrow executes the Turing machine functional specs table.

6 Church's Thesis and Computable Function

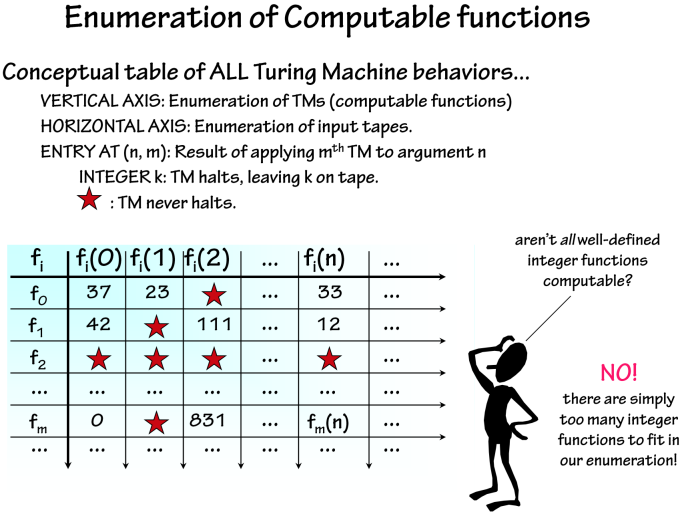
The Church's Thesis: **Every discrete function computable by ANY realisable machine is computable by some Turing machine**.

So, by definition:

$$f(x) \text{ is computable for some k, all x if } f(x) = T_x[x] = f_k(x)$$

(equivalently, f(x) is computable if it is computable by Turing Machine)

Take a look at the figure below to see how we can enumerate all computable functions (represented each by a physical Turing Machines) as the column, and all types of input tapes as the rows, and the cells is the output of applying a specific Turing Machine on a specific input tape:



f_i	$f_i(0)$	$f_i(1)$	$f_i(2)$	...	$f_i(n)$	...
f_0	37	23	★	...	33	...
f_1	42	★	111	...	12	...
f_2	★	★	★	...	★	...
...	...	...	...	...	...	...
f_m	0	★	831	...	$f_m(n)$	...
...	...	...	...	...	...	...

Figure 4

There are also uncomputable function, for example the Halt Function. Google it and find out what it is.

[1] Recall previous section on how you read the tape number x, and in previous section $x = 51$ example was given

7 Universal Function

$$U(k, j) = T_k[j]$$

Explanation:

1. The universal function is a model of general purpose computer
2. Imagine a device with many physical Turing machines inside it, labeled as $T_1, T_2, ...T_k$.... This will be such a big device.
3. Remember, each Turing machine has its functional specification table, and the data "tape"
4. For example, you have Turing machines like Figure 5 below in your lab:

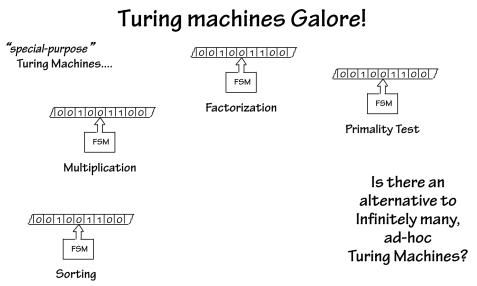


Figure 5

5. So back to the universal function above, k is the input, basically a program to tell us which of computable functions T_k we want to compute,
6. j is the data tape T_k is going to perform at
7. T_u, our universal machine, interprets the data, k and j emulating its program k process on the data, in other words, to emulate the behavior of the T_k (kth Turing Machine) to operate on the data j
8. So with T_u, we no longer need to make so many physical Turing Machines to perform the necessary function, because T_u can emulate the behavior of any Turing Machines
9. T_u is basically our computer, it can read our codes and perform it just as intended. We can just write the code k, and execute it on T_u. For example, we want to multiply a data, factorize them, and then sort them. We no longer need to build three different physical system to multiply, factorize, and split. We just need our one computer to do all the three different tasks.
10. **The Computer Science Revolution**: Imagine building a machine whose input and output is another physical machine, we need combine and compile them to make a new physical machine with different specs. For example, we

want to make a physical hardware that can perform physical sorting using a method like insertion sort, or merge sort. This is very complex to do. We need to replace these ”hardwares” with a coded description of that piece of hardware (which is software), then its easy to cut, change, mix, and modify them around. This is the universal Turing Machine: a coded description of a piece of hardware. This means to move from hardware paradigm into data paradigm. People are no longer spending their time sitting in workshops making physical systems but they are now sitting in front of computers writing codes. Programs can easily input and output other programs.