Math 29: Turing Machines

April 6th, 2022

1 A Different Model of Computing

Register machines were not the original machine model of computation that led to the development of modern hardware. There are many other variations that all happen to be equivalent. We shall discuss one more.

The **tape** for a Turing machine is an infinitely long array, each cell of which contains a 0, a 1, or a $*$ (blank). The **head** for a Turing machine is a pointer which at any given time points to exactly one cell, can read the symbol currently located in that cell, and can move to the adjacent cell on either the left or right. All Turing machines have access to a tape and a head, although the contents of the tape need not start out the same between different machines. When a Turing machine is started, there will be be finitely many 0's and 1's with the head pointing to the leftmost of these non-blank cells. All other cells will be blank.

An **instruction** for a Turing machine is a quadruple

$$\langle \texttt{state}, \texttt{read}, \texttt{act}, \texttt{new\_state} \rangle$$

Each Turing machine has finitely many possible `state` terms. Often the textbook will use $q_0, q_1, q_2, \ldots, q_k$ to represent these states, but you are free to use whatever naming convention you prefer so long as it does not conflict with the other symbols. When the machine is currently in `state`, then the instruction controls what the current behavior of the machine is whenever the cell the head is pointing at contains the symbol `read`, which can be exactly one of 1, 0, or $*$. We will think of natural numbers in binary when thinking about Turing machines. For given `state` and `read` values, there is at most one instruction for the machine to perform.

`act` contains an instruction for the machine to perform. Valid instructions are 0, 1, $*$, L, and R. The first three tell the machine to write the corresponding symbol into the current cell, L tells the machine to move the head one cell to the left, and R similarly tells it to move the head one cell to the right. Finally,

`new_state` is the state that the machine should enter once it finishes its instruction.

A **Turing machine** is a collection of finitely many instructions and a starting state, with optional conditions on the format of the input string. For example, suppose our Turing machine starts in state q_0 and contains the instructions $\langle q_0, 0, R, q_1\rangle$ and $\langle q_0, 1, 0, q_0\rangle$, taking in a consecutive input string. Whenever this machine is in state q_0, it reads the current symbol. If the current symbol is a 0, the head moves right. If it is a 1, then it replaces the 1 with a 0. The machine has no other states and no instruction for when it encounters a blank. A Turing machine **halts** when it has no instructions to perform for its current state and symbol, so this example machine zeroes the tape from left to right until it encounters a blank, then halts.

Example 1. *There is a Turing machine which doubles the input number on the tape then halts.*

Proof: Think about decimal expansions: if we want to multiply a number by ten, we simply add a 0 to the end. The same applies for doubling a number in binary. Therefore, we need a Turing machine which moves right until it encounters a blank, overwrites that blank with a 0, then halts.
Consider the Turing machine made up of the following instructions which starts in state q_0:

1. $\langle q_0, 0, R, q_0\rangle$
2. $\langle q_0, 1, R, q_0\rangle$
3. $\langle q_0, *, 0, q_1\rangle$

Then regardless of the non-blank cells on the tape, the machine moves to the right. Once it encounters a blank, it adds a 0 to the end of the tape, then enters state q_1. Since there are no instructions for q_1, the machine halts. Left on the tape is the original sequence of 1's and 0's with a 0 appended, which corresponds to double the original number in binary.

A function is said to be **(Turing) computable** if there is a Turing machine such that, given n (in binary) starting on the input tape, when the machine is run it will halt with $f(n)$ (also in binary) on the tape. The previous example shows that the function $f(n) = 2n$ is computable.

Lemma 2. *For a fixed m, The two's complement of n in m is computable.*

Proof: Given a number $0 < n < 2^m$, n is represented in binary with up to m bits, and its two's complement in m is the number k such that $n + k = 2^m$. If $n = 0$, then it is its own two's complement. Recall that

$$\Sigma_{i=0}^{m-1} 2^i = 2^m - 1$$

Therefore, in binary $2^m - 1$ is the number with m-many 1's. Therefore, $n + (k - 1) = 2^m - 1$. Thus, to compute k, we need to flip the bits of n then add 1.
We will consider the Turing machine with the following instructions, which starts in state q_0 and accepts as input a consecutive string of length m.

1. $\langle q_0, 0, 1, q_1 \rangle$
2. $\langle q_0, 1, 0, q_1 \rangle$
3. $\langle q_1, 1, R, q_0 \rangle$
4. $\langle q_1, 0, R, q_0 \rangle$
5. $\langle q_0, *, L, q_2 \rangle$
6. $\langle q_2, 1, 0, q_3 \rangle$
7. $\langle q_2, 0, 1, q_3 \rangle$
8. $\langle q_3, 0, L, q_2 \rangle$

In state q_0, it flips any non-blank value and enters state q_1. In state q_1, it travels right when the cell is not blank. These states combine to flip all of the bits across the consecutive input block. On the last non-blank cell, state q_1 will move right to the first blank cell and enter state q_0. On state q_0 when the read value is empty, we move back into the non-blank cells and enter state q_2. In state q_2, we try to add 1. We do so by flipping the bit (since addition by 1 modulo 2 always adds 1), then entering state q_3. If q_3 sees a 0 in the current cell, then we need to carry the 1 by moving left and returning to state q_2. If q_3 sees a 1, then we are finished and the program halts.

As in the case with register machines, a set is Turing computable if either its characteristic or principle function is Turing computable.

Lemma 3. *The set of even numbers is computable.*

Proof: To check if a number is even in binary, we simply need to check that the last bit is 0. We can do this with the following module, assuming the input is a consecutive string of non-blank cells:

- $\langle q_0, 0, R, q_0 \rangle$
- $\langle q_0, 1, R, q_0 \rangle$
- $\langle q_0, *, L, q_1 \rangle$

This module moves right until it reaches the end of the number, then moves left to position the head on the last bit. We enter state q_1 to move into the next phase of the computation.

We now check the bit. If it is 1, then we need to output 0 because the number is odd. If it is 0, then we need to output 1 because the number is even. So we must flip the bit, then blank the rest of the tape before halting. We can do so with the following module:

- $\langle q_1, 1, 0, q_2 \rangle$
- $\langle q_1, 0, 1, q_2 \rangle$
- $\langle q_2, 0, L, q_3 \rangle$
- $\langle q_2, 1, L, q_3 \rangle$
- $\langle q_3, 0, *, q_4 \rangle$
- $\langle q_3, 1, *, q_4 \rangle$
- $\langle q_4, *, L, q_3 \rangle$

State q_1 flips the bit, then enters state q_2, which moves the head left one cell. State q_3 blanks the cell, and then state q_4 moves left. These two states alternate until state q_4 moves left and enters state q_3, but the cell is already blank, at which point the machine halts. (Note that nothing requires the head to point to the output, but we could make this happen by moving right until we find a non-blank cell and halting.)

Lemma 4. *The set of naturals divisible by eight is computable.*

Proof. Homework 2 Question 3. □

Lemma 5. *The set of powers of four is computable.*

Proof. Homework 2 Question 4. □

The differences between Turing machines and register machines make some things easier to compute in one format or the other. As the terminology suggests, though, these two notions of computability are actually the same: given a register machine, we can write a Turing machine that simulates it, and given a Turing machine, we can write a register machine that simulates it.

We will talk about this on Friday, and more details will be done via guided problems on Homework 2. Using these facts, we will be able to use whichever definition of computability is most useful for the given problem. For example, register machines are generally easier for dealing with arithmetic problems, while Turing machines excel at string problems or problems easily represented in binary.