Lecture 29

Disconnect between the Turing Machine representation and actual computers: sequential tape memory vs RAM.

Today we consider a Turing Machine model that is more like an actual computer: **Random Access Turing Machine.**

Infinite number of words in memory, numbered T[0], T[1], . . .
Finite number of registers, R_0, R_1, ... R_k
Program counter κ.

Each register and each word of memory (tape square) can hold an arbitrary integer.

Machine acts as dictated by a *fixed* program – analogous to (fixed) transition function.

The program is a sequence of instructions selected from the following table:

Instruction	Semantics
read j	$R_0 = T[R_j]$
write j	$T[Rj] = R_0$
store j	$R_j = R_0$
load j	$R_0 = R_j$
load =c	$R_0 = c$
add j	$R_0 = R_0 + R_j$
add =c	$R_0 = R_0 + c$
sub j	$R_0 = R_0 - R_j$
sub =c	$R_0 = R_0 - c$
jump s	$\kappa = s$
jpos s	if $R_0 > 0$, then $\kappa = s$
jzero s	if $R_0 == 0$, then $\kappa = s$
halt	$\kappa = 0$

Note that:
Initial register values are 0.
Program counter starts at 1.
R_0 is special – the accumulator.
Instructions go up by 1, unless otherwise specified by a jump.

Example.

```
x = 0;
while (x < 10) {
        <stuff>
        x++;
}
```

can be written as the following sequence, assuming that x is stored in register 1:

1. load 1
2. sub =10
3. jzero 9
4. <stuff>
5. load 1
6. add =1
7. store 1
8. jump 2

More formally,

Def. A Random Access Turing Machine is a pair $M = (k, \Pi)$, where

$k > 0$ is the number of registers, and

$\Pi = (\pi_1, \pi_2, \ldots, \pi_n)$ is the program (i.e., a finite sequence of instructions.

How do we handle i/o in a Random Access Turing Machine? In the spirit of standard Turing Machines: input is a sequence of symbols in memory.

We will assume that symbols are encoded by integers, and that the blank symbol always has value 0.

So, if $\Sigma = \{\#, a, b\}$, then we might assign #=0, a=1, b=2.

Then the input abba would be represented in the first four words of memory as 1,2,2,1.

A configuration of a Random Access TM is a k+2-tuple, where k is the number of registers of the machine. More specifically,

$(\kappa, R_0, R_1, R_2, \ldots, R_{k-1}, T)$, where

k is the current value of the program counter,
$R_0 - R_{k-1}$ are the values of the k registers, and
T describes the contents of memory.

T is a list of the words in memory that are non-zero. That is, it is a set of elements from $N \times (Z - \{0\})$. The first item in each pair gives the memory address; the second item gives the value stored there.

Need notions of **deciding** a language; **semi-deciding** a language; **computing** a function:

In deciding a language, we'll say that M accepts if it reaches a halt configuration with $R_0 = 1$; it rejects if it reaches a halt configuration with $R_0 = 0$. (A halt configuration is one in which the program counter is 0.)

It's not surprising that the Random Access Turing Machine is at least as powerful as a standard Turing Machine.

Let $M = (K, \Sigma, \delta, s, H)$ be a standard Turing Machine.
We can simulate M with M^1, a Random Access Turing Machine:

We'll use one register of M^1 to keep track of M's read/write head on the tape.
Each state's transitions can be simulated by a sequence of instructions.

For example, let $\Sigma = \{\#, a, b, >\}$, and let

$\delta (q, \#) = (p, \rightarrow)$
$\delta (q, a) = (p, \leftarrow)$
$\delta (q, b) = (r, \#)$
$\delta (q, >) = (s, \rightarrow)$, where > is the left edge marker.

We assign each symbol in the alphabet a numeric value:
= 0, a = 1, b = 2, > = 3

The transitions above can then be simulated by the instructions:

If T[n] == 0, then n = n+1; goto p;
If T[n] == 1, n = n-1; goto p;

If T[n] == 2, then T[n] = 0, goto r;
If T[n] == 3, then n = n + 1; goto s;

In these instructions n is the contents of R_1, for example.
goto p means go to the beginning of the instruction sequence for state p.

What's somewhat more surprising (and somewhat more difficult to show) is that any Random Access Turing Machine can be simulated by a regular Turing Machine.

Let M be a Random Access Turing Machine. We'll design M^1 to simulate M.

M^1 will have k+3 tapes, where k is the number of registers of M:
Tape 1 holds input.
Tape 2 holds words of RAM with non-zero values:
    #(0,v_0)(1,v_1)....(i,v_i)#, where v_i is contents of ith word of RAM.
    Each i is represented in binary.
Next k tapes hold contents of k registers.
Last tape is for scratch work.

Instruction numbers will be encoded in state: Each instruction has one set of states for simulation. When done with one, move to the next appropriate set of states. That is, if M has p instructions, then K_1, K_2, ..., K_p are p distinct sets of states used in their simulation.

For example, if the instruction is "read j", search on tape 2 for (j, ...);
If find it, then copy onto the tape representing R_0; if not found, copy 0 into R_0.