Lecture 25

Homework #25: 4.1.8, 4.1.9, 4.1.10

We’d like to be able to create more complex Turing Machines than those we’ve seen so far.

Let’s think about creating a library of simple/useful Turing Machines. We can then use these as subroutines in bigger, more interesting, more complex Turing Machines.

We begin by defining some basic (useful) machines. We can then start talking about how we can combine them.

Most basic: Symbol-writing and head-moving machines.

1. There are $|\Sigma| - 1$ symbol-writing machines. Each writes a specified symbol in the current tape square and then halts. We do not have a symbol-writing machine for the special left-end-of-tape marker.

   The TM to write the symbol $a$ is $M_a = (K, \Sigma, \delta, s, H)$, where $\delta(s,b) = (h,a)$ for all $b \in \Sigma - \{>\}$.
   $K = \{s, h\}$, $H = \{h\}$

   We’ll call $M_a$ "a".

2. There are two basic head moving machines, one that goes left and halts and another that goes right and halts.

   $M_\leftarrow = (K, \Sigma, \delta, s, H)$,
   where $\delta(s,b) = (h, \leftarrow)$ for all $b \in \Sigma - \{>\}$.
   $K = \{s, h\}$, $H = \{h\}$

   We’ll call $M_\leftarrow$ "L" and $M_\rightarrow$ "R".
Now that we have some very simple useful machines to work with, we can think about how to combine them to make bigger Turing Machines. That is, we can think about the mechanics of doing the combining.

- Let $M_1$ be a TM.
- Let $M_2$ be another TM that "calls" $M_1$ as a subroutine.

Operation:
- $M_2$ does some stuff, including making sure that input is ready for $M_1$ (can think of it as sending the right parameters to $M_1$).

\[ M_1's \text{vision of the tape} \]
\[ \begin{array}{cccc} \_ & \_ & \_ & \_ \end{array} \]
\[ \begin{array}{cccc} \# & \_ & \_ & \# \end{array} \]
\[ \_ & \_ & \_ & \_ \]
\[ \_ & \_ & \_ & \_ \]
\[ r/w \text{head positioned for M1} \]
\[ \text{other M2 stuff} \]
\[ \text{input to M1} \]

- Now $M_1$ does its thing.
- Control goes back to $M_2$.

* One more thing we need to assume is that the machines we're combining don't share states. But this is easy because states can be renamed.

Now we can consider the way we'll describe combined machines:

\[ > M_1 \rightarrow M_2 \]

means start $M_1$ and run it till it halts; then run $M_2$ from the position where you just finished.
means run M1. When M1 finishes, if the symbol being scanned is a, then run M2; if it's b, run M3.

Let M1 = (K1, Σ, δ1, s1, H1)
M2 = (K2, Σ, δ2, s2, H2)
M3 = (K3, Σ, δ3, s3, H3)

M, the composite TM, is (K, Σ, δ, s, H), where

K = K1 ∪ K2 ∪ K3
s = s1
H = H2 ∪ H3

For each σ ∈ Σ and q ∈ K-H, δ(q,σ) is defined as follows:
(1) δ(q,σ) = δ1(q,σ), if q ∈ K1-H1.
(2) δ(q,σ) = δ2(q,σ), if q ∈ K2-H2.
(3) δ(q,σ) = δ3(q,σ), if q ∈ K3-H3.
(4) δ(q,a) = (s2,a), if q ∈ H1.
(5) δ(q,b) = (s3,b), if q ∈ H1.
(6) δ(q,b) = (h,b), otherwise, h ∈ H.

And now for some more examples of interesting basic machines:

Ex. Move right until blank: R#

if Σ = {a, b, c}, then R# is

Note that this will always move to the right at least once.
And here are more ways to represent complex machines. The first three are different ways to do the following: move left; if the symbol being scanned is not #, then move left again. Note that I’m using #’ to denote “not #”

The last two are different ways to say “move left twice.”

And now some more TMs that can be used as building blocks for more complex machines:

**Move right/left to x**
[We’ve already seen R#]

- >L # finds 1st # to left of current position
- >R # finds 1st non-blank to right of current position
- >L # finds 1st non-blank to left of current position