Turing-Machine Theory

◆ One purpose of the theory of Turing machines is to prove that certain specific languages have no algorithm.
◆ Start with a language about Turing machines themselves.
◆ Reductions are used to prove more common questions undecidable.

Why Turing Machines?

◆ Why not deal with C programs or something like that?
◆ Answer: You can, but it is easier to prove things about TM’s, because they are so simple.
  ◆ And yet they are as powerful as any computer.
  ◆ More so, in fact, since they have infinite memory.

Then Why Not Finite-State Machines to Model Computers?

◆ In principle, you could, but it is not instructive.
◆ Programming models don’t build in a limit on memory.
◆ In practice, you can go to Fry’s and buy another disk.
◆ But finite automata vital at the chip level (model-checking).

Turing-Machine Formalism

◆ A TM is described by:
  1. A finite set of states (Q, typically).
  2. An input alphabet (Σ, typically).
  3. A tape alphabet (Γ, typically; contains Σ).
  4. A transition function (δ, typically).
  5. A start state (q₀, in Q, typically).
  6. A blank symbol (B, in Γ—Σ, typically).
    ◆ All tape except for the input is blank initially.
  7. A set of final states (F ⊆ Q, typically).

Conventions

◆ a, b, ... are input symbols.
◆ ..., X, Y, Z are tape symbols.
◆ ..., w, x, y, z are strings of input symbols.
◆ α, β, ... are strings of tape symbols.
The Transition Function

- Takes two arguments:
  1. A state, in \( Q \).
  2. A tape symbol in \( \Gamma \).
- \( \delta(q, Z) \) is either undefined or a triple of the form \( (p, Y, D) \).
  - \( p \) is a state.
  - \( Y \) is the new tape symbol.
  - \( D \) is a direction, L or R.

Actions of the PDA

- If \( \delta(q, Z) = (p, Y, D) \) then, in state \( q \), scanning \( Z \) under its tape head, the TM:
  1. Changes the state to \( p \).
  2. Replaces \( Z \) by \( Y \) on the tape.
  3. Moves the head one square in direction \( D \).
     - \( D = L \): move left; \( D = R \): move right.
- If \( \delta(q, Z) \) undefined then the TM halts.

Example: Turing Machine

- This TM scans its input right, looking for a 1.
- If it finds one, it changes it to a 0, goes to final state \( f \), and halts.
- If it reaches a blank, it changes it to a 1 and moves left.
- Net effect?

Example: Turing Machine – (2)

- States = \{q, f\}, \( q \) is start state, \( F=\{f\} \).
- Input symbols = \{0, 1\}.
- Tape symbols = \{0, 1, B\}.
- \( \delta(q, 0) = (q, 0, R) \).
- \( \delta(q, 1) = (f, 0, R) \).
- \( \delta(q, B) = (q, 1, L) \).

Simulation of TM

\( \delta(q, 0) = (q, 0, R) \)
\( \delta(q, 1) = (f, 0, R) \)
\( \delta(q, B) = (q, 1, L) \)

\[ \ldots B B 0 0 B B \ldots \]
Simulation of TM
\[ \delta(q, 0) = (q, 0, R) \]
\[ \delta(q, 1) = (f, 0, R) \]
\[ \delta(q, B) = (q, 1, L) \]

Simulation of TM
\[ \delta(q, 0) = (q, 0, R) \]
\[ \delta(q, 1) = (f, 0, R) \]
\[ \delta(q, B) = (q, 1, L) \]

Simulation of TM
\[ \delta(q, 0) = (q, 0, R) \]
\[ \delta(q, 1) = (f, 0, R) \]
\[ \delta(q, B) = (q, 1, L) \]

Simulation of TM
\[ \delta(q, 0) = (q, 0, R) \]
\[ \delta(q, 1) = (f, 0, R) \]
\[ \delta(q, B) = (q, 1, L) \]

Example: Turing Machine
- States = \{q (start), f (final)\}.
- Input symbols = \{0, 1\}.
- Tape symbols = \{0, 1, B\}.
- \[ \delta(q, 0) = (q, 0, R) \].
- \[ \delta(q, 1) = (f, 0, R) \].
- \[ \delta(q, B) = (q, 1, L) \].
- Does it accept every string?

Instantaneous Descriptions of a Turing Machine
- Initially, a TM has a tape consisting of a string of input symbols surrounded by an infinity of blanks in both directions.
- The TM is in the start state, and the head is at the leftmost input symbol.
TM ID’s – (2)

◆ An ID is a string $\alpha q \beta$, where $\alpha \beta$ is the tape between the leftmost and rightmost nonblanks (inclusive).
◆ The state $q$ is immediately to the left of the tape symbol scanned.
◆ If $q$ is at the right end, it is scanning $B$.
   ▶ If $q$ is scanning a $B$ at the left end, then consecutive $B$'s at and to the right of $q$ are part of $\alpha$.

Formal Definition of Moves

1. If $\delta(q, Z) = (p, Y, R)$, then
   ◆ $\alpha q Z \beta \vdash \alpha Y p \beta$
   ◆ At left end, $Z$ is the blank, so $\alpha q \vdash \alpha Y p$
2. If $\delta(q, Z) = (p, Y, L)$, then
   ◆ For any $X$, $\alpha X q Z \beta \vdash \alpha p X Y \beta$
   ◆ At right end, move to blank: $q Z \beta \vdash p B Y \beta$

Languages of a TM

◆ A TM defines a language by final state, as usual.
◆ $L(M) = \{w | q_0 \vdash w \vdash^* I, \text{where I is an ID with a final state}\}$.
◆ Or, a TM can accept a language by halting.
◆ $H(M) = \{w | q_0 \vdash w \vdash^* I, \text{and there is no move possible from ID I}\}$.

Equivalence of Accepting and Halting

1. If $L = L(M)$, then there is a TM $M'$ such that $L = H(M')$.
2. If $L = H(M)$, then there is a TM $M''$ such that $L = L(M'')$.

Proof of 1: Acceptance -> Halting

◆ Modify $M$ to become $M'$ as follows:
1. For each accepting state of $M$, remove all transitions (if any), so $M'$ halts in that state.
2. Avoid having $M'$ accidentally halt (crash).
   ◆ Introduce a new state $s$, which runs to the right forever; that is $\delta(s, X) = (s, X, R)$ for all symbols $X$.
   ◆ If $q$ is not accepting, and $\delta(q, X)$ is undefined, make $\delta(q, X) = (s, X, R)$.
Proof of 2: Halting -> Acceptance

- Modify M to become M* as follows:
  1. Introduce a new state f, the only accepting state of M*.
  2. f has no moves.
  3. If $\delta(q, X)$ is undefined for any (other) state q and symbol X, define it by $\delta(q, X) = (f, X, R)$.

Recursively Enumerable Languages

- We now see that the classes of languages defined by TM's using final state and halting are the same.
- This class of languages is called the recursively enumerable languages.
  - Why? The term actually predates the Turing machine and refers to another notion of computation of functions.

Recursive Languages

- An "algorithm" is a TM that is guaranteed to halt (on every input) whether or not it accepts.
- If $L = L(M)$ for some TM M that is an algorithm, we say L is a recursive language.
  - Why? Again, don't ask; it is a term with a history.

Example: Recursive Languages

- Every CFL is a recursive language.
  - Use the CYK algorithm.
- Every regular language is a CFL (think of its DFA as a PDA that ignores its stack); therefore every regular language is recursive.
- Almost anything you can think of is recursive.
  - But not all languages are ...