ECS 120 Lesson 15 – Turing Machines, Pt. 1

Oliver Kreylos

Wednesday, May 2nd, 2001

Before we can start investigating the really interesting problems in theoretical computer science, we have to introduce yet another type of machine. As we have seen recently, neither finite automata nor pushdown automata are powerful enough to accept certain types of languages that have practical applications, like the language of syntactically correct `C` programs.

We have found out, however, that pushdown automata are more powerful than finite automata, and that this increase in power is based in the availability of unlimited memory in form of a stack. To further increase the power of our computation models, we will remove the restriction of last-in, first-out access from pushdown automata. The resulting machine, that can access its memory in a random-access fashion, is called a *Turing Machine*. This machine type was first introduced by Alan Turing in 1936.

1 Turing Machines

A Turing Machine is essentially still a finite state machine, but as opposed to the latter, it can write to the input tape (now called *work tape*, and it can move its read/write head arbitrarily over the work tape. Furthermore, the work tape does not only contain the word that is the machine's input, but it has an unlimited number of tape cells to the right. An illustration of a Turing Machine is shown in Figure 1.

A Turing Machine computes by initially being presented with its input written on the work tape, from the leftmost position to the right. All work tape cells to the right of the end of the input tape are filled with a special *blank symbol*. Initially, the read/write head is positioned over the leftmost tape cell. The Turing Machine will then transition inside its state control, guided by

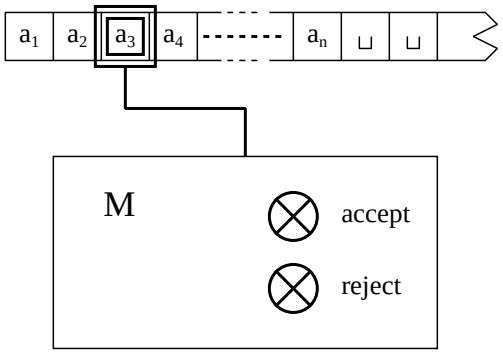

Figure 1: A Turing Machine M consisting of state control, work tape and movable read/write head.

the character located underneath the read/write head, called the *current character*. In each computation step, the machine writes a new character into the tape cell underneath the read/write head, and then moves the head either to the left or to the right. Unlike a finite automaton, a Turing Machine ends its computation by entering either the designated *accept state* or the *reject state*. If the machine never enters one of these states, computation will continue forever and it will never halt.

To summarize, a Turing Machine differs from a finite automaton in the following ways:

- A Turing Machine can write to its input tape (work tape).
- The read/write head can be moved to the left and to the right.
- The work tape is unlimited to the right.
- The machine has two special accept and reject states which take immediate effect.

The major difference between a Turing Machine and a pushdown automaton is that the former can access its work tape in a random-access fashion, whereas the latter can only access its stack in a last-in, first-out manner.

2 Examples on Turing Machine Computation

To get a feel for how Turing Machines work, let us look at some languages that we know are not context-free, and see how we can recognize them using Turing Machines.

2.1 The Language $L_1 := \left\{ \mathtt{0}^n\mathtt{1}^n\mathtt{2}^n \;\middle|\; n \geq 0 \right\}$

The idea for a Turing Machine to accept this language is to sweep the input from left to right, crossing of exactly one `0`, one `1` and one `2` in each sweep. If after one of the sweeps only crossed-out characters remain, the word must initially have consisted of an equal number of zeros, ones and twos.

We will give a high-level description of a Turing Machine that accepts language L_1 in the form of an *algorithm*. Initially, the input word w is written on the work tape, and all cells to the right of the input tape are filled with the blank symbol. The read/write head is positioned at the left end of the tape. The machine now proceeds as follows:

1. If the input word is the empty word, i. e., the current character is the blank symbol, go to the accept state.

2. If the current character is not a zero, go to the reject state. Otherwise, cross it out and move the read/write head to the right until something but a zero or a crossed-out character is encountered.

3. If the current character is not a one, go to the reject state. Otherwise, cross it out and move the read/write head to the right until something but a one or a crossed-out character is encountered.

4. If the current character is not a two, go to the reject state. Otherwise, cross it out and move the read/write head to the right until something but a two or a crossed-out character is encountered.

5. If the current character is not the blank symbol, go to the reject state. Otherwise, move the read/write head back to the left end of the tape. If all characters encountered on the way are crossed-out, move to the accept state.

6. Move the read/write head to the right until the first non-crossed-out character is encountered and repeat from step 2.

2.2 The Language $L_2 := \left\{ 0^{2^n} \mid n \geq 0 \right\}$

This language seems a little more complicated to accept. After all, a Turing Machine is little more than a glorified finite state machine, and we already now that finite state machines cannot perform arithmetics. Thus, we have to find a different way to determine whether a string contains a power of two number of zeros than calculating the exponential directly. This other way lies in a recursive defition of all powers of two:

Base Case If $n = 1$, then n is a power of two.

Inductive Case If n is even, and $n/2$ is a power of two, then n is a power of two.

This observation leads to the following idea: We sweep the input from left to right, and in each sweep we determine if the number of zeros seen was even or odd, or if it was exactly one. While sweeping, we cross out every other zero. If the number of zeros was one, we accept. If the number of zeros was odd, we reject. If the number was even, we have effectively divided the number by two by crossing out every other zero. We now move the read/write head back to the beginning and repeat. In the form of an algorithm, the Turing Machine to accept L_2 works as follows:

1. If the input word is the empty word, i. e., the current character is the blank symbol, go to the reject state.

2. If the current character is not a zero, reject. Otherwise, move the input head one to the right.

3. Scan to the right, until the first non-crossed-out character is encountered. If it is the blank symbol, accept. If it is neither the blank symbol nor a zero, reject.

4. Cross out the current character and scan to the right until the next non-crossed-out character is encountered. If it is the blank symbol, move the read/write head all the way to the left end of the tape and repeat from step 2. If it is neither the blank symbol nor a zero, reject.

5. Move the read/write head one to the right and scan to the right until the next non-crossed-out character is encountered. If it is anything but a zero, reject. Otherwise, repeat from step 4.

3 Formal Definition of Turing Machines

Though the description of Turing Machines by algorithms, as shown in the above sections, is usually sufficient to understand how a Turing Machine works, it is not precise enough to allow us to reason about the behaviour of a specific Turing Machine. For this purpose we need a formal definition.

A Turing Machine M is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where

- Q is a finite set of states,
- Σ is the input alphabet, where $\sqcup \notin \Sigma$,
- Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$,
- $\delta: Q \times \Gamma \to Q \times \Gamma \times \{\text{L}, \text{R}\}$ is the transition function,
- $q_0 \in Q$ is the start state,
- $q_{\text{accept}} \in Q$ is the accept state, and
- $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{accept}} \neq q_{\text{reject}}$.

4 Turing Machines as State Diagrams

As for finite automata, a Turing Machine's behaviour is determined by its transition function. To specify the transition function, we can draw a transition diagram similar to those for finite automata or pushdown automata. The machine's states will be drawn as nodes, and the transition function will be denoted by arrows going from state to state, labeled with the read input character, the character written to the tape, and the direction the tape head should move. Figure 2 shows how the arrows are labeled for different types of transitions.

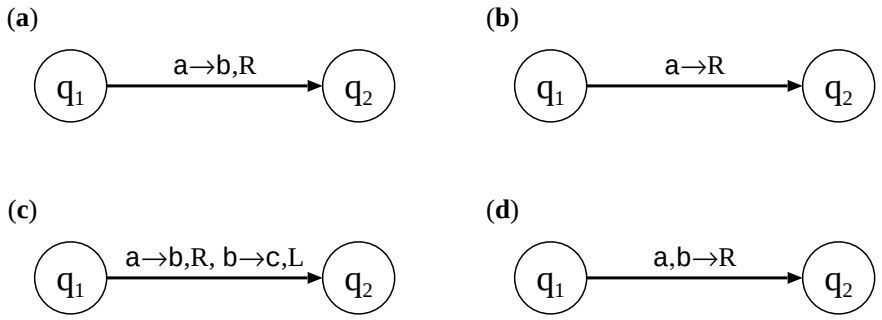

Figure 2: Notations and shorthands for transition diagrams of Turing Machines. (**a**) Label for a transition $\delta(q_1, \mathtt{a}) = (q_2, \mathtt{b}, \mathrm{R})$ (**b**) Label for a transition $\delta(q_1, \mathtt{a}) = (q_2, \mathtt{a}, \mathrm{R})$ (the character `a` is not changed) (**c**) Label for two transitions $\delta(q_1, \mathtt{a}) = (q_2, \mathtt{b}, \mathrm{R})$ and $\delta(q_1, \mathtt{b}) = (q_2, \mathtt{c}, \mathrm{L})$ (**d**) Label for two transitions $\delta(q_1, \mathtt{a}) = (q_2, \mathtt{a}, \mathrm{R})$ and $\delta(q_1, \mathtt{b}) = (q_2, \mathtt{b}, \mathrm{R})$ (the characters `a` and `b` are not changed).