Lecture 12:
Coding Schemes for Turing Machines,
Universal Turing Machines,
The Halting Problem

Lecture Outline

- A Coding System for Turing Machines
- A Non-Phrase Structure Language
- Universal Turing Machines
- Acceptable and Decidable Languages
- The Halting Problem

Reading

Chapter 3.4 - 3.6 of Brookshear

Chapter 9 and 12 of Martin, 2nd ed.

Chapter 6.4 of Revesz

Chapter 29 of Cohen

Chapter 8 of Hopcroft and Ullman

A Coding System for Turing Machines

- It is possible to completely describe a Turing machine M with alphabet Σ and tape symbols $\Sigma \cup \{\Delta\}$ by using a coding scheme consisting entirely of 0's and 1's.

- Proceed as follows:

  1. Arrange M's states in a list with the start state first and halt state second and other states following. This permits each state of M to be assigned a number, based on position in the list. Represent the j-th state of M by a string of j 0's.
  2. Arrange the symbols of Σ in a list. Represent the left transition symbol L by 0, the right transition symbol R by 00, the first symbol in the list of symbols in Σ by 000, and in general the jth symbol in this list by $(j+2)$ 0's.
  3. Represent the blank by the empty string
  4. Represent a transition $\delta(p, x) = (q, y)$ as a string of 0's and 1's of the form

     $$p\text{'s code } 1\ x\text{'s code } 1\ q\text{'s code } 1\ y\text{'s code}$$

     E.g. using 1's to delimit strings of 0's, the transition $\delta(\iota, x) = (h, R)$ would be represented as

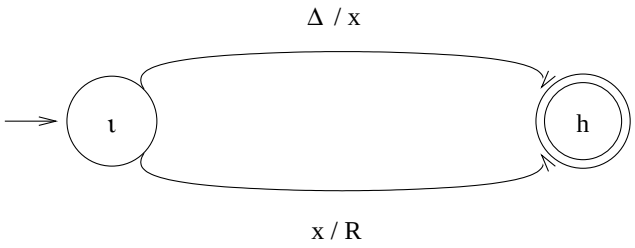
     $$\overbrace{0}^{\iota}\,1\,\overbrace{000}^{x}\,1\,\overbrace{00}^{h}\,1\,\overbrace{00}^{R}$$

     Here we have assumed x is coded as 000 – i.e as the first symbol in the list of symbols of Σ.
  5. represent a Turing machine as a sequence of coded transitions with an extra 1 at the beginning and a 1 at the end and a 1 separating each pair of transitions.

A Coding System for Turing Machines (cont)

- In order to regularise the procedure of coding any particular TM the following additional conventions are adopted:
  1. transitions are listed ordered by the state from which they originate, i.e. transitions originating from state 0 are listed first, then those from state 000 (recall state 00 is the halt state), state 0000, and so on.
  2. transitions originating from the same state are listed ordered by the symbol required on the current tape cell, i.e. the transition requiring the blank is listed first, that requiring the symbol whose code is 000 second (recall L and R are coded 0 and 00), that requiring the symbol whose code is 0000 third, and so on.

- A simple TM



which can be represented according to our coding scheme as:

$$101100100010100010010 01$$

i.e.,

$$\overbrace{\overbrace{1\,\overbrace{0}^{\iota}\,1\,\overbrace{}^{\Delta}\,1\,\overbrace{00}^{h}\,1\,\overbrace{000}^{x}}^{Transition1}\,\overbrace{1\,\overbrace{0}^{\iota}\,1\,\overbrace{000}^{x}\,1\,\overbrace{00}^{h}\,1\,\overbrace{00}^{R}}^{Transition2}\,1}$$

A Non-Phrase Structure Language

- We can now use this coding scheme to demonstrate the existence of non-phrase structure languages.

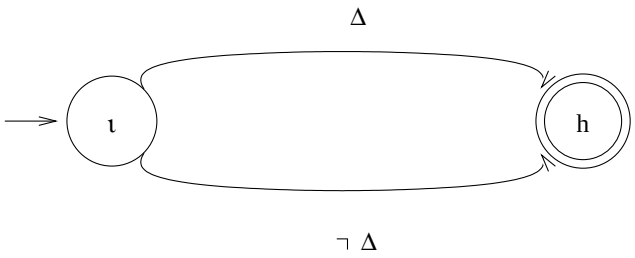
- We have seen how every TM with alphabet Σ and tape symbols $\Sigma \cup \{\Delta\}$ can be represented as a string of 0's and 1's.

  These strings can be interpreted as nonnegative binary numbers.

  Thus, if we started at 0 and counted upwards in binary we would eventually arrive at the binary number representing any given TM with tape symbols $\Sigma \cup \{\Delta\}$.

  Of course many of the binary numbers we would encounter are *not* valid representations of any machine.

  If we agree to associate with each of these non-well-formed representations some trivial TM such as:



  then we can define a function from **N** onto the Turing Machines with alphabet Σ and tape symbols $\Sigma \cup \{\Delta\}$.

  We represent the TM which is the value of this function at the integer i by $M(i)$.

- Based on this function we can construct another function, from Σ^* onto the set of TM's with alphabet Σ and tape symbols $\Sigma \cup \{\Delta\}$.

  For any string w in Σ^* we map w onto the TM M_i where i is the length $|w|$ of w. We denote the machine associated thus with w by $M_{|w|}$ or more simply M_w.

A Non-Phrase Structure Language (cont)

- For any string w in Σ^* the symbols of w are in the alphabet of M_w.

  Thus, we can supply w to M_w as an input string and see whether or not M_w halts.

  We define L_0 as the subset of Σ^* $\{w \mid M_w$ does not accept $w\}$. That is, L_0 consists of all those strings whose corresponding machines *do not* accept them.

- We now argue that L_0 is not Turing-acceptable.

- Suppose L_0 is Turing-acceptable. Then there is some TM with alphabet Σ and tape symbols $\Sigma \cup \{\Delta\}$ that accepts it (by the argument above showing Δ is the only tape symbol needed in addition to those in Σ).

  Since every TM is M_w for some string w in Σ^* it follows that the TM that accepts L_0 must be M_{w_0} for some w_0 in Σ^*.

  Therefore, $L_0 = L(M_{w_0})$.

- Is w_0 in $L(M_{w_0})$?

  If $w_0 \in L(M_{w_0})$ then since L_0 is defined as $\{w \mid M_w$ does not accept $w\}$ it follows that $w_0 \notin L_0$.

  If $w_0 \notin L(M_{w_0})$ then since L_0 is defined as $\{w \mid M_w$ does not accept $w\}$ it follows that $w_0 \in L_0$.

- But $L_0 = L(M_{w_0})$. Thus $w_0 \in L(M_{w_0})$ implies $w_0 \notin L(M_{w_0})$ and vice versa.

- This is contradictory. Therefore, our supposition that L_0 is Turing-acceptable must be false, i.e. L_0 is not Turing-acceptable.

Universal Turing Machines

- A **universal Turing machine** is a Turing machine that is able to simulate the behaviour of any other Turing machine.

- A universal Turing machine (UTM) executes a program stored on its tape.

  It may be thought of as a 'programmable' TM – an abstract analogue of today's general purpose digital computers which fetch and execute stored programs.

- A program for a UTM is just a coded TM which performs the task which we want the UTM to perform.

  That is, to get a UTM to do some particular task, we define a TM to do that task, code it according to our earlier coding scheme, then supply this coded TM as a program to our UTM which decodes it and executes the instructions as if it were the specific TM from which the program was derived.

- In order to use a UTM not only must we be able to present a coded TM as a program to it, we must also be able to present it with the input the specific TM would be asked to process.

  Recall that symbols in the alphabet Σ may be coded as strings of three or more 0's – we introduced this as part of our coding scheme for transitions.

  So, we can code an input string as a string of substrings of 0's each representing a symbol in the input string, separated by 1's acting as symbol delimiters.

  We also start and end the entire coded input string with a 1.

Universal Turing Machines (cont)

- Thus our UTM will be presented with a string of 0's and 1's representing a specific TM for it to simulate, plus a string of 0's and 1's representing the input the machine is to process.

  We adopt the convention of placing these strings on the UTM's input tape as follows:

  1. the leftmost cell remains blank
  2. starting in the second cell is the coded version of the specific TM to be simulated (the 'program')
  3. immediately following the coded version of the TM to be simulated is the coded input

- Note that no confusion can occur between 'instructions' (transitions) and 'data' (input) because the last transition ends with a 1 and the beginning of the data is marked with a 1.

  Since transitions must begin with a 0, any attempt to interpret the data as yet another instruction must fail.

- For example here is a coded TM with input data, as they might be presented to a UTM:

$$\overbrace{1\,\underbrace{011001000}_{Transition1}\,1\,\underbrace{01000100100}_{Transition2}\,1}^{CodedMachine}\quad\overbrace{1\,\underbrace{000}_{Symbol1}\,1\,\underbrace{00000}_{Symbol2}\,1\,\underbrace{0000}_{Symbol3}\,1}^{CodedData}$$

Universal Turing Machines (cont)

- Given such a representation of a coded machine and coded data , there are numerous ways a UTM could be designed.

  One approach is to build a machine with three tapes, the first for the program, the input data, and any output, the second as a work tape for manipulating input and the third for keeping track of the current state of the simulated machine.

  Brookshear p. 182-184 provides a complete specification for a UTM using this approach.

- Such a UTM proceeds as follows:

  1. find the beginning of the coded input string and copy onto tape 2
  2. place the code for the initial state on tape 3
  3. search the coded transitions (the 'program') on tape 1 until an applicable transition is found.
  4. simulate the transition on tape 2
  5. update the current state code on tape 3 to be the new simulated state
  6. if the simulated state becomes the halt state, erase tape 1, copy tape 2 onto tape 1, position the tape head on tape 1 where the tape head was on tape 2 when the halt state was reached, and halt.

- While this a three-tape machine, the result about multiple tape TM's ensures us that a one-tape machine can be constructed which will simulate the three-tape machine.

Acceptable and Decidable Languages

- Using a UTM we can construct a TM which accepts the complement of the language L_0 which we showed above to be not Turing acceptable.

  Recall L_0 was defined as $\{w \mid M_w \text{ does not accept } w\}$.

  The complement of L_0 therefore is the language $\{w \mid M_w \text{ accepts } w\}$.

- First, construct a TM M_{pre} which preprocesses an input string $w \in \Sigma^*$ as follows:

  1. generates a coded representation of the machine M_w – recall M_w will either be the default machine if the binary representation of $|w|$ is not a valid TM representation, or the binary representation of $|w|$ otherwise.
  2. places the result on its tape followed by the coded representation of w.

- Suppose we denote our UTM by M_U. A machine which accepts the complement of L_0 is the composite machine:

  $$\rightarrow M_{pre} \rightarrow M_U$$

  Given an input w this machine effectively applies M_w to w and halts if and only if M_w would halt when given w.

  Therefore this machine accepts $\{w \mid M_w \text{ accepts } w\}$, i.e. it accepts the complement of L_0.

Acceptable and Decidable Languages (cont)

- We have just shown that there are languages which may be Turing-acceptable, but whose complements are not.

- One effect of this is that there are languages L for which
  1. we can build a TM which when given strings w will respond by writing a Y on its tape if $w \in L$
  2. we cannot build a TM which when given strings w will respond by writing a Y on its tape if $w \in L$ and a N on its tape if $w \notin L$.

- Languages whose strings are accepted by some TM are called *Turing acceptable*.

  Languages whose strings are accepted by some TM which is also capable of rejecting strings not in the language, e.g. by writing a N message, are called **Turing-decidable** languages.

  (so all Turing-decidable languages are Turing acceptable, but the converse is not true).

- This distinction is important practice. Suppose we are working with a language L which is known to be merely *Turing acceptable*. Now suppose we are given a string w to test for membership in L. If the machine fails to halt after any finite amount of processing time then we cannot know whether $w \in L$ and we have just not processed long enough or whether $w \notin L$.

- One class of languages which is Turing-decidable is the class of context-free languages.

- The Turing-acceptable languages are also called the **recursively enumerable** languages.

- The Turing-decidable languages are also called the **recursive** languages.

The Halting Problem

- Recall that any TM can be coded as a string of 0's and 1's.

- We denote the coded version of a TM M by $\rho(M)$.

- Restricting ourselves to machines with alphabet $\{0, 1\}$ and tape symbols $\{0, 1, \Delta\}$, we can consider what M does if $\rho(M)$ (itself in coded form) is supplied to it as input.

- If a machine M halts with $\rho(M)$ as input we call it **self-terminating**.

  Otherwise M is **non-self-terminating**.

  Note that every TM M with alphabet $\{0, 1\}$ and tape symbols $\{0, 1, \Delta\}$ is either self-terminating or non-self-terminating.

- Define a language L_h to be $\{\rho(M) \mid M \text{ is self-terminating}\}$.

  I.e. L_h is the language consisting of coded representations of self-terminating Turing Machines.

- Is L_h Turing-decidable ?

- L_h is Turing-decidable if it is possible to design a TM which can determine whether or not a given string of 0's and 1's is the coded representation of a Turing Machine which halts when applied to itself.

  Hence this problem is called the **halting problem**.

The Halting Problem (cont)

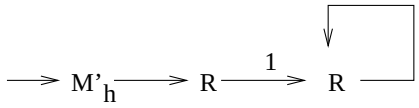
L_h is **not** Turing-decidable. We show this as follows:

- Suppose it is Turing-decidable. Then there must exist a TM M_h which decides it.

- Define a TM M'_h just like M_h except that M'_h halts with a 1 on its tape when M_h halts with a Y and M'_h halts with a 0 on its tape when M_h halts with a N.

- Note that the tape symbols of M'_h need only be in $\{0, 1, \Delta\}$ (as we saw above).

- Using M'_h we can specify another machine M_0 with tape symbols $\{0, 1, \Delta\}$ as:



  M_0 halts only if M'_h halts with output 0 – otherwise it loops forever.

The Halting Problem (cont)

- Is M_0 self-terminating ?

- Suppose M_0 is self-terminating.

  Since M'_h halts with output 1 when given the coded representation of a self-terminating machine, M'_h must halt with output 1 when given $\rho(M_0)$ as input.

  But then M_0 would **not** halt with input $\rho(M_0)$ since it loops forever if M'_h halts with output 1. I.e. it is not self-terminating.

- Suppose M_0 is not self-terminating.

  Then M'_h must halt with output 0 when given $\rho(M_0)$ as input.

  But in this case M_0 would halt when given $\rho(M_0)$ as input. I.e. it is a self-terminating machine.

- Either M_0 is or is not self-terminating. Hence we have arrived at a contradiction and so our assumption that L_h was Turing-decidable must be false.

- Thus, we have now identified two non-Turing-decidable languages – L_h and L_0.