Lecture 31

Universal Machines and Diagonalization

A Universal Turing Machine

Now we come to a crucial observation about the power of Turing machines: there exist Turing machines that can simulate other Turing machines whose descriptions are presented as part of the input. There is nothing mysterious about this; it is the same as writing a LISP interpreter in LISP.

First we need to fix a reasonable encoding scheme for Turing machines over the alphabet $\{0,1\}$. This encoding scheme should be simple enough that all the data associated with a machine M—the set of states, the transition function, the input and tape alphabets, the endmarker, the blank symbol, and the start, accept, and reject states—can be determined easily by another machine reading the encoded description of M. For example, if the string begins with the prefix

$$0^n10^m10^k10^s10^t10^r10^u10^v1,$$

this might indicate that the machine has n states represented by the numbers 0 to $n-1$; it has m tape symbols represented by the numbers 0 to $m-1$, of which the first k represent input symbols; the start, accept, and reject states are s, t, and r, respectively; and the endmarker and blank symbol are u and v, respectively. The remainder of the string can consist

of a sequence of substrings specifying the transitions in δ. For example, the substring

$$0^p10^a10^q10^b10$$

might indicate that δ contains the transition

$$((p,a),\ (q,b,L)),$$

the direction to move the head encoded by the final digit. The exact details of the encoding scheme are not important. The only requirements are that it should be easy to interpret and able to encode all Turing machines up to isomorphism.

Once we have a suitable encoding of Turing machines, we can construct a *universal Turing machine* U such that

$$L(U) \stackrel{\text{def}}{=} \{M\#x \mid x \in L(M)\}.$$

In other words, presented with (an encoding over $\{0,1\}$ of) a Turing machine M and (an encoding over $\{0,1\}$ of) a string x over M's input alphabet, the machine U accepts $M\#x$ iff M accepts x.[1] The symbol # is just a symbol in U's input alphabet other than 0 or 1 used to delimit M and x.

The machine U first checks its input $M\#x$ to make sure that M is a valid encoding of a Turing machine and x is a valid encoding of a string over M's input alphabet. If not, it immediately rejects.

If the encodings of M and x are valid, the machine U does a step-by-step simulation of M. This might work as follows. The tape of U is partitioned into three tracks. The description of M is copied to the top track and the string x to the middle track. The middle track will be used to hold the simulated contents of M's tape. The bottom track will be used to remember the current state of M and the current position of M's read/write head. The machine U then simulates M on input x one step at a time, shuttling back and forth between the description of M on its top track and the simulated contents of M's tape on the middle track. In each step, it updates M's state and simulated tape contents as dictated by M's transition function, which U can read from the description of M. If ever M halts and accepts or halts and rejects, then U does the same.

As we have observed, the string x over the input alphabet of M and its encoding over the input alphabet of U are two different things, since the two machines may have different input alphabets. If the input alphabet of

[1] Note that we are using the metasymbol M for both a Turing machine and its encoding over $\{0,1\}$ and the metasymbol x for both a string over M's input alphabet and its encoding over $\{0,1\}$. This is for notational convenience.

M is bigger than that of U, then each symbol of x must be encoded as a string of symbols over U's input alphabet. Also, the tape alphabet of M may be bigger than that of U, in which case each symbol of M's tape alphabet must be encoded as a string of symbols over U's tape alphabet. In general, each step of M may require many steps of U to simulate.

Diagonalization

We now show how to use a universal Turing machine in conjunction with a technique called ***diagonalization*** to prove that the halting and membership problems for Turing machines are undecidable. In other words, the sets

$$\mathrm{HP} \stackrel{\text{def}}{=} \{M\#x \mid M \text{ halts on } x\},$$

$$\mathrm{MP} \stackrel{\text{def}}{=} \{M\#x \mid x \in L(M)\}$$

are not recursive.

The technique of diagonalization was first used by Cantor at the end of the nineteenth century to show that there does not exist a one-to-one correspondence between the natural numbers $\mathbb{N}$ and its *power set*

$$2^{\mathbb{N}} = \{A \mid A \subseteq \mathbb{N}\},$$

the set of all subsets of $\mathbb{N}$. In fact, there does not even exist a function

$$f : \mathbb{N} \to 2^{\mathbb{N}}$$

that is onto. Here is how Cantor's argument went.

Suppose (for a contradiction) that such an onto function f did exist. Consider an infinite two-dimensional matrix indexed along the top by the natural numbers $0, 1, 2, \ldots$ and down the left by the sets $f(0), f(1), f(2), \ldots$. Fill in the matrix by placing a 1 in position i, j if j is in the set $f(i)$ and 0 if $j \notin f(i)$.

	0	1	2	3	4	5	6	7	8	9	$\cdots$
$f(0)$	1	0	0	1	1	0	1	0	1	1	
$f(1)$	0	0	1	1	0	1	1	0	0	1	
$f(2)$	0	1	1	0	0	0	1	1	0	1	
$f(3)$	0	1	0	1	1	0	1	1	0	0	
$f(4)$	1	0	1	0	0	1	0	0	1	1	$\cdots$
$f(5)$	1	0	1	1	0	1	1	1	0	1	
$f(6)$	0	0	1	0	1	1	0	0	1	1	
$f(7)$	1	1	1	0	1	1	1	0	1	0	
$f(8)$	0	0	1	0	0	0	0	1	1	0	
$f(9)$	1	1	0	0	1	0	0	1	0	0	
$\vdots$					$\vdots$						$\ddots$

The ith row of the matrix is a bit string describing the set $f(i)$. For example, in the above picture, $f(0) = \{0, 3, 4, 6, 8, 9, \ldots\}$ and $f(1) = \{2, 3, 5, 6, 9, \ldots\}$. By our (soon to be proved fallacious) assumption that f is onto, every subset of $\mathbb{N}$ appears as a row of this matrix.

But we can construct a new set that does not appear in the list by complementing the main diagonal of the matrix (hence the term *diagonalization*). Look at the infinite bit string down the main diagonal (in this example, $1011010010\cdots$) and take its Boolean complement (in this example, $0100101101\cdots$). This new bit string represents a set B (in this example, $B = \{1, 4, 6, 7, 9, \ldots\}$). But the set B does not appear anywhere in the list down the left side of the matrix, since it differs from every $f(i)$ on at least one element, namely i. This is a contradiction, since every subset of $\mathbb{N}$ was supposed to occur as a row of the matrix, by our assumption that f was onto.

This argument works not only for the natural numbers $\mathbb{N}$, but for any set A whatsoever. Suppose (for a contradiction) there existed an onto function from A to its power set:

$$f : A \to 2^A.$$

Let

$$B = \{x \in A \mid x \notin f(x)\}$$

(this is the formal way of *complementing the diagonal*). Then $B \subseteq A$. Since f is onto, there must exist $y \in A$ such that $f(y) = B$. Now we ask whether $y \in f(y)$ and discover a contradiction:

$$\begin{aligned} y \in f(y) &\iff y \in B && \text{since } B = f(y) \\ &\iff y \notin f(y) && \text{definition of } B. \end{aligned}$$

Thus no such f can exist.

Undecidability of the Halting Problem

We have discussed how to encode descriptions of Turing machines as strings in $\{0,1\}^*$ so that these descriptions can be read and simulated by a universal Turing machine U. The machine U takes as input an encoding of a Turing machine M and a string x and simulates M on input x, and

- halts and accepts if M halts and accepts x,
- halts and rejects if M halts and rejects x, and
- loops if M loops on x.

The machine U doesn't do any fancy analysis on the machine M to try to determine whether or not it will halt. It just blindly simulates M step by step. If M doesn't halt on x, then U will just go on happily simulating M forever.

It is natural to ask whether we can do better than just a blind simulation. Might there be some way to analyze M to determine in advance, before doing the simulation, whether M would eventually halt on x? If U could say for sure in advance that M would not halt on x, then it could skip the simulation and save itself a lot of useless work. On the other hand, if U could ascertain that M *would* eventually halt on x, then it could go ahead with the simulation to determine whether M accepts or rejects. We could then build a machine U' that takes as input an encoding of a Turing machine M and a string x, and

- halts and accepts if M halts and accepts x,
- halts and rejects if M halts and rejects x, and
- halts and rejects if M loops on x.

This would say that $L(U') = L(U) = \text{MP}$ is a recursive set.

Unfortunately, this is not possible in general. There are certainly machines for which it is possible to determine halting by some heuristic or other: machines for which the start state is the accept state, for example. However, there is no general method that gives the right answer for all machines.

We can prove this using Cantor's diagonalization technique. For $x \in \{0,1\}^*$, let M_x be the Turing machine with input alphabet $\{0,1\}$ whose encoding over $\{0,1\}^*$ is x. (If x is not a legal description of a TM with input alphabet $\{0,1\}^*$ according to our encoding scheme, we take M_x to be some arbitrary but fixed TM with input alphabet $\{0,1\}$, say a trivial TM with one state that immediately halts.) In this way we get a list

$$M_\epsilon, M_0, M_1, M_{00}, M_{01}, M_{10}, M_{11}, M_{100}, M_{101}, \ldots \tag{31.1}$$

containing all possible Turing machines with input alphabet $\{0,1\}$ indexed by strings in $\{0,1\}^*$. We make sure that the encoding scheme is simple enough that a universal machine can determine M_x from x for the purpose of simulation.

Now consider an infinite two-dimensional matrix indexed along the top by strings in $\{0,1\}^*$ and down the left by TMs in the list (31.1). The matrix

contains an H in position x, y if M_x halts on input y and an L if M_x loops on input y.

	ϵ	0	1	00	01	10	11	000	001	010	$\cdots$
M_ϵ	H	L	L	H	H	L	H	L	H	H	
M_0	L	L	H	H	L	H	H	L	L	H	
M_1	L	H	H	L	L	L	H	H	L	H	
M_{00}	L	H	L	H	H	L	H	H	L	L	
M_{01}	H	L	H	L	L	H	L	L	H	H	$\cdots$
M_{10}	H	L	H	H	L	H	H	H	L	H	
M_{11}	L	L	H	L	H	H	L	L	H	H	
M_{000}	H	H	H	L	H	H	H	L	H	L	
M_{001}	L	L	H	L	L	L	L	H	H	L	
M_{010}	H	H	L	L	H	L	L	H	L	L	
$\vdots$					$\vdots$						$\ddots$

The xth row of the matrix describes for each input string y whether or not M_x halts on y. For example, in the above picture, M_ϵ halts on inputs $\epsilon, 00, 01, 11, 001, 010, \ldots$ and does not halt on inputs $0, 1, 10, 000, \ldots$.

Suppose (for a contradiction) that there existed a *total* machine K accepting the set HP; that is, a machine that for any given x and y could determine the x, yth entry of the above table in finite time. Thus on input $M\#x$,

- K halts and accepts if M halts on x, and
- K halts and rejects if M loops on x.

Consider a machine N that on input $x \in \{0,1\}^*$

(i) constructs M_x from x and writes $M_x\#x$ on its tape;

(ii) runs K on input $M_x\#x$, accepting if K rejects and going into a trivial loop if K accepts.

Note that N is essentially complementing the diagonal of the above matrix. Then for any $x \in \{0,1\}^*$,

$$\begin{aligned} N \text{ halts on } x &\iff K \text{ rejects } M_x\#x && \text{definition of } N \\ &\iff M_x \text{ loops on } x && \text{assumption about } K. \end{aligned}$$

This says that N's behavior is different from every M_x on at least one string, namely x. But the list (31.1) was supposed to contain all Turing machines over the input alphabet $\{0,1\}$, including N. This is a contradiction. □

The fallacious assumption that led to the contradiction was that it was possible to determine the entries of the matrix effectively; in other words,

that there existed a Turing machine K that given M and x could determine in a finite time whether or not M halts on x.

One can always simulate a given machine on a given input. If the machine ever halts, then we will know this eventually, and we can stop the simulation and say that it halted; but if not, there is no way in general to stop after a finite time and say for certain that it will never halt.

Undecidability of the Membership Problem

The membership problem is also undecidable. We can show this by *reducing* the halting problem to it. In other words, we show that if there were a way to decide membership in general, we could use this as a subroutine to decide halting in general. But we just showed above that halting is undecidable, so membership must be undecidable too.

Here is how we would use a total TM that decides membership as a subroutine to decide halting. Given a machine M and input x, suppose we wanted to find out whether M halts on x. Build a new machine N that is exactly like M, except that it accepts whenever M would either accept or reject. The machine N can be constructed from M simply by adding a new accept state and making the old accept and reject states transfer to this new accept state. Then for all x, N accepts x iff M halts on x. The membership problem for N and x (asking whether $x \in L(N)$) is therefore the same as the halting problem for M and x (asking whether M halts on x). If the membership problem were decidable, then we could decide whether M halts on x by constructing N and asking whether $x \in L(N)$. But we have shown above that the halting problem is undecidable, therefore the membership problem must also be undecidable.