Turing machines (part 2)

Uncomputability

Uncomputability: overview

- As we shall see, every Turing machine can be encoded into a list of natural numbers.
- So the set of Turing machines is enumerable.
- As we have seen, the set of functions $N \longrightarrow N$ is not enumerable.
- So there must be functions $N \longrightarrow N$ which are not Turing-computable.
- We shall find **concrete examples** of non-computable functions.

Enumerating TM's

	0	1
q_1	$W_1 q_1$	$L q_2$
q_2	$W_1 q_2$	$L q_3$
q_3	$W_1 q_3$	

- Recall that we can present a Turing machine as a transition table.
- The table can be presented as a list of quadruples: $q_1 0 W_1 q_1,\ q_1 1 L q_2,\ q_2 0 W_1 q_2,\ q_2 1 L q_3,\ q_3 0 W_1 q_3$.

Enumerating TM's

- The sets Q, $\Sigma = \{0, 1\}$, and $\{W_0, W_1, L, R\}$ are finite.
- So the set $Q \times \Sigma \times \{W_0, W_1, L, R\} \times Q$ of quadruples is finite, and in particular enumerable.
- So the set $(Q \times \Sigma \times \{W_0, W_1, L, R\} \times Q)^*$ of finite lists of quadruples is enumerable.
- Because every TM can be represented by such a list, the set of TM's is enumerable.

Enumerating Turing-computable functions

- We have an enumeration of TM's:

$$M_1, M_2, M_3, \ldots$$

- Letting $f_i : N \to N$ be the function computed by M_i, we have an enumeration of the set of Turing-computable functions:

$$f_1, f_2, f_3, \ldots$$

Functions not Turing-computable

So there must be functions $N \rightarrow N$ that are not Turing-computable, for otherwise $f_1, f_2, f_3, \ldots$ would be an enumeration of $N \rightarrow N$, which is impossible because of the diagonalization argument.

The diagonal function

Let $f_1, f_2, f_3, \ldots$ be an enumeration of Turing machines. The **diagonal function** d is defined as follows:

$$d(n) = \begin{cases} \perp & \text{if } f_n(n) \text{ is defined,} \\ 1 & \text{otherwise} \end{cases}$$

(Recall that we write $\perp$ for "undefined".)

Link with the Java example

- Recall the Java program `test2` that takes a string `program`, “runs `program` on itself”, prints “Hello World!” if `program` does **not** print “Hello World!”, and “Whatever” otherwise.
- The diagonal function d is similar: it takes a natural number n that represents a Turing machine, “runs n on itself”, and “does the opposite” of the result.

Uncomputability of d

- As we have seen, the Java program `test2` cannot be Java-computable.
- Similarly, the diagonal function d cannot be Turing-computable (see lecture for proof).

The halting function

The **halting function** is defined as follows:

$$h(n,k) = \begin{cases} 2 & \text{if } M_n \text{ halts on input } k \\ 1 & \text{otherwise} \end{cases}$$

Naïve attempt at computing $h(n, k)$

- Run the n-th TM, M_n, on input k.
- If the computation halts, then go into an infinite loop.
- If the computation does not halt, return 1.
- But how long do we wait for the computation to halt?

Self-halting

The **self-halting function** is defined by $s(n) = h(n, n)$.

Proposition. The self-halting function s is not Turing-computable.

Proof. See lecture.

Uncomputability of the halting function

Corollary. The halting function h is not Turing-computable.

Proof. See lecture.

Summary

- The main result is that the halting function is not Turing-computable.
- Informally, this means that there is no TM that takes as inputs a code n for a TM and a number m and decides whether M_n halts on m or not.