1 “Modern version of the Church-Turing Thesis” (in scare quotes)

The usual way to state the Church-Turing Thesis is that any precisely described algorithmic process can be modelled using a Turing machine. For the modern generation, who grew up believing computers can implement every precisely described algorithm, we have a “Modern version of the Church-Turing Thesis.”

Claim 1. Anything computable by a digital computer can be computed by a Turing machine.

Features of a digital computer:

- There is no distinction between instructions and data
- There is a finite control that updates with instructions/data
- The data is stored in words
- There are program counters and registers.

At this point there is much hand-waving that a Turing machine could actually do all this. But surely they can.

2 Computable functions

In what follows we will be dealing with two sort of functions, \( f : \mathbb{N} \to \mathbb{N} \) and \( f : \Sigma^* \to \Sigma^* \), where \( \Sigma = \{0, 1\} \), and taking star of a set indicates the set of (finite) strings made up of that set. To associate a function, \( f \), mapping natural numbers to natural numbers with a Turing machine, we need to specify a convention for encoding natural numbers as strings that could appear on a tape head of a Turing machine. The two conventions are unary notation, in which the natural number \( i \) is encoded by \( 0^i \), a string of \( i \) many 0’s, and binary notation, in which the natural number \( i \) is written as
its binary expansion. We adopt the convention that such binary numbers are written with 1 in the most significant bit. This makes it such that there is exactly one binary representation for every natural number (i.e. \(0010 = 10\), but the LHS is not allowed, and the RHS is allowed).

We are now prepared to say what it means for a Turing machine \(M\) to compute a function \(f\).

**Definition 1.** We define what it means for a Turing machine \(M\) to compute \(v\) on input \(\omega\). Let \(M\) be a Turing machine with the following input conventions:

- \(M\) begins with \(\omega \in \Sigma^*\) on its tape, with the tape head at the first symbol of \(\omega\).
- The rest of the tape consists of the empty symbol.

The Turing machine \(M\) has the following output conventions:

- \(M\) halts in the state \(q_H\), and when it does, the tape head is at the leftmost symbol of some maximal length string \(v \in \Sigma^*\).

When these conditions are fulfilled, we say that \(M\) compute \(v\) on input \(\omega\). This is the same as writing \(M(\omega) = v\). We may write \(M(\omega) \downarrow = v\), or just \(M(\omega) \downarrow\). If \(M(\omega)\) never halts, then we write \(M(\omega) \uparrow\).

Furthermore, when \(M\) operates on unary representations of numbers, we require \(\omega = 0^i\), \(\Sigma = \{0\}\), so \(v = 0^v\). When \(M\) operates on binary representations of numbers, \(\omega \in \{0, 1\}^*\) either starting with 1 or the empty string.

**Definition 2.** Given a Turing machine \(M\) and function \(f: \Sigma^* \rightarrow \Sigma^*\), if \(\forall \omega \in \Sigma^*, M(\omega) \uparrow = f(\omega)\) then we say \(M\) computes \(f\).

**Definition 3.** Let \(f\) be a function, if there exists a Turing machine \(M\) such that \(M\) computes \(f\), then we say that \(f\) is computable or recursive.

**Definition 4.** Let \(f\) be a partial function (in other words, a function \(f\) where \(\text{dom}(f) \subseteq \Sigma^*\) and \(\text{range}(f) \subseteq \Sigma^*\)). We say \(M\) computes \(f\) when

\[(\forall \omega)(\omega \in \text{dom}(f) \implies M(\omega) \downarrow = f(\omega)) \text{ and } \omega \notin \text{dom}(f) \implies M(\omega) \uparrow)\]

**Definition 5.** If \(f\) is a partial function, and there exists an \(M\) that computes \(f\), then we say that \(f\) is partial recursive or partial computable.
3 Relations

Definition 6. A relation or predicate is a set $R \subseteq \Sigma^*$.

Definition 7. A Turing machine $M$ decides a relation $R$ if $M$ has two halting states, $q_Y$ and $q_N$ (accepting and rejecting) and for all $\omega \in \Sigma^*$, $\omega \in R$ has $M(\omega)$ halt in $q_Y$ and $\omega \notin R$ has $M(\omega)$ halt in $q_N$. We say that “$M$ accepts $\omega$” and “$M$ rejects $\omega$,” respectively.

Definition 8. A relation $R$ is decidable if there exists a Turing machine $M$ s.t. $M$ decides $R$.

Definition 9. A Turing machine $M$ semidecides $R$ if $\forall \omega (\omega \in R \iff M(\omega) \text{ accepts } \omega)$.

Definition 10. A relation $R$ is semidecidable if there exists a Turing machine $M$ that semidecides $R$.

Definition 11. A Turing machine $M$ enumerates $R \subseteq \Sigma^*$ when $M$ has a “pause” state $q_P$, and when $M$ is run on blank input, it periodically enters the pause state, with the output of $M$ at this point being an element of $R$. This gives a sequence $\omega_1, \omega_2, \ldots$ of output values, which may be finite or infinite, and may contain duplicates. Further impose the condition that $R = \{\omega_1, \omega_2, \ldots\}$. In other words, every element in $R$ eventually gets enumerated.

Definition 12. $R$ is recursively enumerable (r.e.) or computably enumerable (c.e) if there is some Turing machine $M$ that enumerates $R$.

Theorem 1. $R$ is semidecidable iff $R$ is recursively enumerable.

Proof. Suppose $R$ is r.e.; we want to show that $R$ is semidecidable. In other words, given a Turing machine $M_1$ that enumerates $R$, we want a Turing machine $M_2$ that semidecides $R$.

Algorithm for $M_2$

- Input $\omega \in \Sigma^*$.
- Run $M_1$ (on a blank input tape)
- Every time $M_1$ goes into $q_P$, and outputs $\omega_1$, check whether or not $\omega = \omega_1$. If so, then halt in $q_Y$. Otherwise, keep running $M_1$.

For the other direction: Suppose $M_3$ semidecides $R$, we want to give an algorithm for $M_4$ that enumerates $R$.

Algorithm for $M_4$
• Loop over $i = 1, 2, 3, \ldots$
  
  – For each $\omega \in \Sigma^*$, $|\omega| \leq i$,
    * Run $M_3$ on input $\omega$ for up to $i$ steps.
    * If $M_3$ enters its $q_Y$ in this process, then enter $q_P$ and output $\omega$.
  
  – End for

• End loop

Discussion about the backward direction construction above: We might think to try and construct an algorithm for $M_4$ simply be running $M_3$ on every input. The problem with this is that $M_3$ is not guaranteed to halt on every input. We avoid this problem by only running $M_3$ for a limited number of steps. This way, if $M_3$ does halt on an input, we will eventually discover it, but we are not bound to simply run $M_3$ indefinitely. One way to visualize what is going on here is to imagine that we are running several Turing machines running $M_3$ on different inputs in parallel. When one of them halts, we announce it and enter the pause state. But one of them not halting does not break the whole process.

Final point: All of these points generalize to $k$-ary functions and relations.