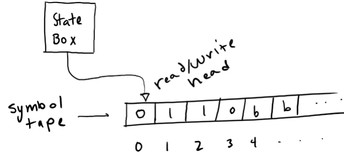
What is the Recursion Theorem?

Turing Machines

What is a Turing Machine? Formally, a Turing machine is a 7-tuple $(Q, \Gamma, b, \Sigma, q_0, F, \delta)$ where Q is a finite set of "states," Γ is a finite set of "tape symbols," $b \in \Gamma$ is the "blank symbol," $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of "input symbols," $q_0 \in Q$ is the "initial state," and $F \subseteq Q$ is the set of "final states." The most important part is the "transition function," $\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$. This function describes the operation of the machine: given the current state of the machine and the current symbol under the read/write head, what state will the machine take in the next time step, what symbol will the read/write head write over the symbol it's currently seeing in the tape, and what direction should the read/write head move on the tape?



What does a Turing Machine do? For our purposes, we will think of a Turing Machine as starting in state q_0 with the read/write head at position 0 of the symbol tape and with an "input string" (some word over Σ, an element of Σ^*) printed on the symbol tape starting at position 0, and with the blank symbol written on all of the other positions on the symbol tape. If a Turing Machine enters some state in F, its operation halts (the transition function's domain doesn't include states in F) and we say that the Turing Machine "returns" the string written on the tape starting at position 0 and up until the first position on the tape which has a symbol not in Σ (the blank symbol b, for example). This string (also an element of Σ^*) is also called the "output string."

Notation: If a Turing Machine M enters a final state (or "halts") when given input string x, we write $M(x)\downarrow$ (otherwise we write $M(x)\uparrow$). If this happens and M has returned the string y, we write $M(x)\downarrow= y$.

What else can a Turing Machine do? There is some Turing Machine M which can simulate any other Turing Machine M', receiving as its input string an encoding of M' into tape symbols, along with an input string for M' (note that we fix here some specific encoding of Turing Machines that will be used throughout). Another thing which is computable by Turing Machine is the ability to check whether or not its input string is a well-formed encoding of some other Turing Machine (returning 0 or 1, or some other encoding of False or True).

How many Turing Machines are there? We henceforth restrict our investigation to Turing Machines operating over the tape alphabet $\Gamma = \{0, 1, b\}$ and input alphabet $\Sigma = \{0, 1\}$. We also restrict Q, the set of states, to consist of elements $q_0, q_1, q_2, \ldots$, essentially just removing the freedom to name states as we please. This class of Turing Machines is enough to consider since any Turing Machine which uses more symbols in its tape or input alphabet can be simulated by a Turing Machine of this class, simply assigning each symbol a bit-string (i.e. if the input alphabet is $\{\alpha, \beta, \gamma, \delta\}$, we can let the string 00 represent α, 01 for β, 10 for γ and 11 for delta). Thus, since each set in the description for a Turing Machine is finite, there are only countably many Turing Machines, meaning we can index the set of Turing Machines by $\mathbb{N}$.

Indexing the Turing Machines: For our purposes, we will index the Turing Machines by the number associated with their encoding in binary: the first Turing Machine is the one whose binary encoding, when read as a number (i.e. 101 is the binary encoding of 5) is lowest. We write M_j for the jth Turing Machine under this index. As a side note, any bijective function $f : \mathbb{N} \to \mathbb{N}$ which is computable (i.e. "total recursive," more about this soon) gives rise to another suitable index, i.e. $M_{f(j)}$ can be the jth Turing Machine and the rest of our construction will work fine.

One more important capability: Using our indexing system (or any other computable index), there is a Turing Machine which can, given some input number j, find and output the encoding for M_j. This is acheivable simply by running through each binary word in numerical order, checking to see if the current word is a properly formed Turing Machine encoding, and (if it is) increasing some stored variable until that variable hits j, at which point the binary word will be returned. Similarly, it is possible for some Turing Machine to return the index of a Turing Machine whose encoding is given as input.

The Recursion Theorem

Definitions: A "partial function" is a function $f : \mathbb{N} \to \mathbb{N} \cup \{\bot\}$ (think of $\bot$ as "undefined"). A partial function f is called a "partial recursive" function if it is computed by some Turing Machine M_j, i.e. whenever $f(x) = y$, if $y \in \mathbb{N}$ we have $M_j(x) \downarrow = y$ and if $y = \bot$ we have $M_j(x) \uparrow$. Every Turing Machine computes some partial recursive function, and we write φ_j to denote the partial recursive function computed by M_j. A "total recursive" function is a partial recursive function whose range does not include $\bot$ (the Turing Machine which computes it halts on every input).

Notation: Since $|\mathbb{N}^k| = |\mathbb{N}|$, let $(x_1, x_2, \dots, x_k) \mapsto \langle x_1, x_2, \dots, x_k \rangle$ be your favorite computable injection from $\mathbb{N}^k$ into $\mathbb{N}$, perhaps $\langle x_1, x_2, x_3, \dots, x_k \rangle = 2^{x_1} 3^{x_2} 5^{x_3} \cdots p_k^{x_k}$. Using this, when we'd like to think of functions having multiple inputs, we really only need a single-input function.

The $S_{m,n}$ Theorem: Let f be a partial recursive function. Then there exists a total recursive function σ so that for all $i, j \in \mathbb{N}$, $\varphi_{\sigma(i)}(j) = f(\langle i, j \rangle)$.

Proof. Given i, let M be a Turing Machine which, given input j, encodes $\langle i, j \rangle$ on the tape and then simulates the machine which computes f, using $\langle i, j \rangle$ as the input for the simulated machine. Let $\sigma(i)$ be the index of M. ∎

The Recursion Theorem: Let σ be a total recursive function. Then there is some index n so that $\varphi_n = \varphi_{\sigma(n)}$.

Proof. Consider a partial recursive function f which has $f(\langle i, j \rangle) = \varphi_{\sigma(\varphi_i(i))}(j)$ (if $\varphi_i(i) = \bot$ we say the whole expression is $\bot$). By the $S_{m,n}$ Theorem, there is a total recursive function g which has $\varphi_{g(i)}(j) = f(\langle i, j \rangle)$. Thus we have $\varphi_{g(i)} = \varphi_{\sigma(\varphi_i(i))}$ for every i. Let $n \in \mathbb{N}$ so that M_n computes g. Then $\varphi_{g(n)} = \varphi_{\sigma(\varphi_n(n))} = \varphi_{\sigma(g(n))}$ since $\varphi_n(n) = g(n)$, and so $g(n)$ is the "fixed point index" whose existence was posited. ∎

The Second Recursion Theorem: Let f be a partial recursive function. Then there is an index n so that for all $j \in \mathbb{N}$, $\varphi_n(j) = f(\langle n, j \rangle)$.

Proof. Let σ be a total recursive function so that $\varphi_{\sigma(i)}(j) = f(\langle i, j \rangle)$ for any $i, j \in \mathbb{N}$ (the existence of σ is due to the $S_{m,n}$ theorem). Then let n be the index so that $\varphi_n = \varphi_{\sigma(n)}$ (whose existence is due to the first recursion theorem). We have $\varphi_n(j) = \varphi_{\sigma(n)}(j) = f(\langle n, j \rangle)$ for any j. ∎

Examples of Quines

Lisp

```
((lambda (x)
    (list x (list (quote quote) x)))
 (quote
    (lambda (x)
       (list x (list (quote quote) x)))))
```

Python

```
a = ['print "a =", a', 'for s in a: print s']
print "a =", a
for s in a: print s
```

References & Further Reading

[1] Kleene, Stephen Cole. "Introduction To Metamathematics". 1950.

[2] Turing, Alan Mathison. "Computability and λ-definability". The Journal of Symbolic Logic. December 1937.

[3] The Quine Page. http://www.nyx.net/ gthompso/quine.htm