The recursion theorem

The second theorem, called the *recursion theorem*, states that every total recursive function σ mapping *indices* (integers denoting Turing machines) of partial recursive functions into indices of partial recursive functions has a fixed point x_0 such that $f_{x_0}(y) = f_{\sigma(x_0)}(y)$ for all y. In other words, if we modify all Turing machines in some manner, there is always some Turing machine M_{x_0} for which the modified Turing machine $M_{\sigma(x_0)}$ computes the same function as the unmodified Turing machine. At first this sounds impossible, since we can modify each Turing machine to add 1 to the originally computed function. One is tempted to say that $f(y) + 1 \neq f(y)$. But note that if $f(y)$ is everywhere undefined, then $f(y) + 1$ does equal $f(y)$ for all y.

Theorem 8.18 For any total recursive function σ there exists an x_0 such that $f_{x_0}(x) = f_{\sigma(x_0)}(x)$ for all x.

Proof For each integer i construct a TM that on input x computes $f_i(i)$ and then simulates, by means of a universal TM, the $f_i(i)$th TM on x. Let $g(i)$ be index of the TM so constructed. Thus for all i and x,

$$f_{g(i)}(x) = f_{f_i(i)}(x). \tag{8.3}$$

Observe that $g(i)$ is a total function even if $f_i(i)$ is not defined. Let j be an index of the function σg. That is, j is an integer code for a TM that, given input i, computes $g(i)$ and then applies σ to $g(i)$. Then for $x_0 = g(j)$ we have

$$\begin{aligned} f_{x_0}(x) &= f_{g(j)}(x) \\ &= f_{f_j(j)}(x) && \text{by (8.3)} \\ &= f_{\sigma(g(j))}(x) && \text{since } f_j \text{ is the function } \sigma g \\ &= f_{\sigma(x_0)}(x). \end{aligned}$$

Thus x_0 is a fixed point of the mapping σ. That is, TM x_0 and TM $\sigma(x_0)$ compute the same function. □

Applications of the recursion and S_{mn} theorems

Example 8.9 Let $M_1, M_2, \ldots$ be any enumeration of all Turing machines. We do not require that this enumeration be the "standard" one introduced in Section 8.3, but only that whatever representation is used for a TM, we can by an algorithm convert from that representation to the 7-tuple notation introduced in Section 7.2, and vice versa. Then we can use the recursion theorem to show that for some i, M_i and M_{i+1} both compute the same function.

Let $\sigma(i)$ be the total recursive function defined as follows. Enumerate TM's $M_1, M_2, \ldots$ until one with integer code i as in (8.2) is found. Note that the states of the TM must be considered in all possible orders to see if i is a code for this TM, since in the notation introduced in Section 8.3, the order in which the moves for the various states is written affects the code. Having found that M_j has code i, enumerate one more TM, M_{j+1}, and let $\sigma(i)$ be the code for M_{j+1}. Then the recursion theorem applied to this σ says there is some x_0 for which M_{x_0} and M_{x_0+1} define the same function of one variable.

Example 8.10 Given a formal system F, such as set theory, we can exhibit a Turing machine M such that there is no proof in F that M started on any particular input halts, and no proof that it does not halt. Construct M, a TM computing a two-input function $g(i, j)$, such that

$$g(i,j) = \begin{cases} 1 & \text{if there is a proof in } F \text{ that } f_i(j) \text{ is} \\ & \text{not defined; that is, there is a proof} \\ & \text{that the } i\text{th TM does not halt when given input } j; \\ \text{undefined otherwise.} \end{cases}$$

M enumerates proofs in F in some order, printing 1 if a proof that the ith TM does not halt on input j is found. Further, we may construct M so that if $g(i, j) = 1$, then M halts, and M does not halt otherwise. By the S_{mn}-theorem there exists σ such that

$$f_{\sigma(i)}(j) = g(i, j).$$

By the recursion theorem, we may effectively construct an integer i_0 such that

$$f_{i_0}(j) = f_{\sigma(i_0)}(j) = g(i_0, j).$$

But $g(i_0, j) = 1$, and is therefore defined, if and only if there is a proof in F that $f_{i_0}(j)$ is undefined. Thus if F is consistent (i.e., there cannot be proofs of a statement and its negation), there can be no proof in F that the i_0th TM either halts or does not halt on any particular input j.

8.9 ORACLE COMPUTATIONS

One is tempted to ask what would happen if the emptiness problem, or some other undecidable problem, were decidable? Could we then compute everything? To answer the question we must be careful. If we start out by assuming that the emptiness problem is decidable, we have a contradictory set of assumptions and may conclude anything. We avoid this problem by defining a Turing machine with oracle.

Let A be a language, $A \subseteq \Sigma^*$. A *Turing machine with oracle A* is a single-tape Turing machine with three special states $q_?$, q_y, and q_n. The state $q_?$ is used to ask whether a string is in the set A. When the Turing machine enters state $q_?$ it requests an answer to the question: "Is the string of nonblank symbols to the right of the tape head in A?" The answer is supplied by having the state of the Turing