

Recursive Functions – An Introduction
Martin Davis
Belfer Graduate School of Science
Yeshiva University, New York, New York

In “Automata Theory” ed. E.R. Caianiello
Academic Pres, New York 1966, p. 153-163.

I. *What Is Recursive Function Theory?*

Although recursive function theory has, perhaps, little to do with automata theory, Turing machines are being discussed here, and surely recursive function theory is the *raison d’être* for Turing machines.

Recursive function theory is, in a suitable sense, the theory of computation, or better, of computability. To see what is involved, consider the usual elementary-school algorithm for adding a pair of integers given in decimal notation. For numbers of what size does this algorithm work? The immediate reply, “for numbers of any size”, is certainly not literally correct; e.g., we cannot use the algorithm for adding numbers too large to be written on a blackboard the size of the Pacific Ocean. Nevertheless, everyone feels that he understands what is meant by saying that the algorithm works for all numbers. It is with *computation procedures in this somewhat metaphorical sense* that recursive function theory is concerned.

Since an actual digital computer is a finite object subject to limitations of space and time (of which practical programmers are only too aware), one might at first sight assume that digital computers have little to do with the notion of computation being proposed here. But when one considers the availability of auxiliary storage (tapes, cards, etc.), it is seen that the precise limits of a given machine are not well defined. To summarize, we may say that in contexts where it is proper to think of a computing machine augmented with as much storage as needed, and computing as long as necessary to solve a given problem, we are using the word “computation” in the sense intended in recursive function theory.

A function f defined and with values on the nonnegative integers $0, 1, 2, 3, \dots$ (e. g., $f(x) = x^2$) is called recursive or computable if there is a computing procedure (in the sense indicated) for computing the values of f .¹⁵⁴ This is not, of course, a rigorous definition, because the meaning of the term “computing procedure” has been indicated only in an informal manner. What makes the theory of recursive (or computable) functions possible is that the class of recursive functions (as defined alone) turns out to be *quite* insensitive to the details of the definition given of computing procedure – definitions which vary widely among themselves all lead to the same class!

II. *Turing’s Analysis of Computation*

Here, we shall follow Turing’s analysis of computation as carried out by a human computer, which led him, by the process of simplification by omission of the inessential, to what we now called Turing machines.

A human computer is observed writing symbols on a sheet of paper in a two-dimensional array, e.g.,

$$\begin{array}{r}
32 \\
62 \\
\hline
64 \\
192 \\
\hline
1984
\end{array}$$

Our first simplification is to note that with no loss in computing power, a one-dimensional “strip” of paper would do as well, e.g.,

$$32, 62; 64, 192; 1984.$$

The computer is at any moment of time aware only of the symbols appearing in a small part of the paper. This is really the essential idea in the notion of a Turing machine – that is behavior is influenced by “local” information only.

Our next simplification is to regard as our atomic symbols the data actually “scanned” at a given instant.

Finally, we assume that the Turing machine has a finite number of *internal states* or *configurations* and that the next action of a given Turing machine is entirely determined by its present internal configuration, together with the symbol currently scanned. This next action is taken to be either a replacement of the symbol scanned by another, or motion of one square to the left or to the right, followed, in either case, by transition to a new internal configuration.

The fact that a given Turing machine in state q scanning symbol S replaces S by S' and goes into state q' can be expressed by the “quadruple” $qSS'q'$.¹⁵⁵ Similarly, motions to the left or right are associated with quadruples $qS \leftarrow q'$, $qS \rightarrow q'$. Mathematically, we can simply define a Turing machine to be a finite set of quadruples in which no two quadruples begin with the same pair qS . (This last stipulation is just to avoid contradictory instructions.)

We may think of Turing machines in two ways:

1. As above, the q 's are configurations (of toothpicks and rubber bands – if these are what the machine is made of) of the machine.
2. The q 's are instructions carried out by, an abstract machine which can only overprint or move one square.

The latter point of view was the one taken by Post in his work (independent of Turing's) and recently revived by Hao Wang.

In a rigorous development of the subject, each assertion of the recursiveness of a function should be justified by showing that there is a Turing machine which computes the function. Here we shall content ourselves with indicating heuristically that an algorithm exists for computing the function, relying on the cogency of Turing's analysis to convince ourselves that a corresponding Turing machine exists.

III. Gödel Numbers and a Universal Function

When a Turing machine arrives at a state q , scanning a symbol S , where the machine contains no quadruple beginning qS , the machine will stop, and the computation will be over. However, there may well be arguments for which the machine never stops, but simply continues computing forever. In this case, the machine “computes” a function

whose domain of definition is a subset of the set of natural numbers. Such a function is called *partial recursive* or *partially computable*. An example is $f(x) = 10 - x$, undefined for $x > 10$. Also, we shall speak of recursive or partial recursive functions of more than one argument. It is simply necessary to put all the arguments on the tape initially, separated by punctuation marks.

Let us use integers (written decimally) as subscripts on q and S to yield notations for the various internal states and symbols of a Turing machine. Rewriting the subscripts “on the line” the description of a Turing machine becomes a sequence of the symbols $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, q, S, \leftarrow, \rightarrow$. Taking $q, S, \leftarrow, \rightarrow$ as new digits denoting the numbers 10, 11, 12, and 13, respectively, our description of a Turing machine becomes a number written to the base 14. This number is called the Gödel number of the Turing machine.

156

We now define the important function $\mathcal{A}_z(x)$ as follows:

$\mathcal{A}_z(x)$ is the partial recursive function computed by the Turing machine with Gödel number z , if there is such; otherwise $\mathcal{A}_z(x) = 0$.

We have:

1. For each partial recursive function $f(x)$ there is a z such that

$$f(x) = \mathcal{A}_z(x),$$

i.e., the sequence

$$\mathcal{A}_0(x), \mathcal{A}_1(x), \mathcal{A}_2(x), \dots$$

is an enumeration of all partial recursive functions (of one argument).

Proof. Let z be the Gödel number of a Turing machine which computes $f(x)$.

2. $\mathcal{A}_z(x)$ is a partial recursive function of two arguments.

For, given z and x , to compute $\mathcal{A}_z(x)$, write z to the base 14 and check whether z is a Gödel number of a Turing machine. If not, continue computing forever. If it is, apply the quadruples of that machine to x on its tape. This is an algorithm; hence $\mathcal{A}_z(x)$ is partially computable.

Let U be a Turing machine which computes $\mathcal{A}_z(x)$. Then U is called universal, because by placing a suitable z_0 on its tape it can be made to compute any partial recursive function.

IV. Recursive and Recursively Enumerable Sets

Let

$$\omega_i = \{x \mid \mathcal{A}_i(x) \text{ is defined}\}.$$

Then $\omega_0, \omega_1, \omega_2, \dots$ is an enumeration of all sets which can serve as a domain of definition of partial recursive functions. Such sets are called *recursively enumerable* (re). A set R is recursive if its characteristic function

$$C_R(x) = 1 \text{ if } x \in R; C_R(x) = 0 \text{ if } x \notin R$$

is recursive. Clearly to say that R is recursive is to say that there is an algorithm by means of which we can test a given number to determine whether or not it belongs to R .

Theorem. R is recursive iff R and \bar{R} (i.e., the set of integers not in R) are both re. 157

Proof. If R is recursive, then the functions

$$\begin{aligned} f_1(x) &= 1 \text{ if } x \in R, & f_1(x) &\text{ undefined otherwise} \\ f_2(x) &= 0 \text{ if } x \in \bar{R}, & f_2(x) &\text{ undefined otherwise} \end{aligned}$$

are both partial recursive, as is readily seen by considering the Turing machine which computes $C_R(x)$. Hence, R and \bar{R} are recursive.

Conversely, let R and \bar{R} be recursive so that they are the domains of definition of partial recursive functions computed by Turing machines T_1 and T_2 , respectively. Then, the following is an algorithm for testing whether or not a given integer x belongs to R :

Place x on the tapes of T_1 and T_2 , and let both begin computing. Wait until one of them halts. If it is T_1 , then $x \in R$; if T_2 , then $x \notin R$.

The question is suggested by this theorem: Is there a recursive but nonrecursive set? Set $K = \{i \mid i \in \omega_i\}$, so that K is the set of Gödel numbers of Turing machines which halt when their own Gödel number is initially placed on their tape. Then, we have

Theorem. K is recursive, but \bar{K} is not. So, K is not recursive.

Proof. Since K is the domain of definition of the partial recursive function $\mathcal{A}_i(i)$, K is recursive.

Suppose \bar{K} is recursive so that

$$\bar{K} = \omega_q.$$

Then

$$i \in \bar{K} \Leftrightarrow i \in \omega_q.$$

But

$$i \in \bar{K} \Leftrightarrow i \notin \omega_i.$$

Hence, for every i ,

$$i \in \omega_q \Leftrightarrow i \notin \omega_i.$$

Setting $i = q$.

$$q \in \omega_q \Leftrightarrow q \notin \omega_q.$$

This is a contradiction.

The argument used in this last proof will be recognized at once as an application of Cantor's diagonal method.

V. Gödel's Incompleteness Theorem

Mathematicians are in the habit of proving theorems and communicating their results to others. For the purpose of this communication, a *language* ¹⁵⁸ either natural or artificial must be used. Thus, to each assertion and to each proof must correspond a finite sequence of symbols (i.e., a linguistic expression). To serve its purpose, one must be able to check mechanically, that an alleged proof really is a proof. In practice this last is an unrealized ideal (as everyone who has struggled with "obviouslies" and "clearlies" will readily admit). However, the formal systems of logic constructed by logicians do meet this demand. The content of Gödel's theorem is that if this demand is met, then not all true statements of arithmetic can be proved.

We shall take the following fixed set of mathematical assertions:

$$\begin{aligned} 0 &\in \bar{K} \\ 1 &\in \bar{K} \end{aligned}$$

$$\begin{aligned}
2 &\in \bar{K} \\
3 &\in \bar{K} \\
&\dots \\
&\dots
\end{aligned}$$

Let us suppose that for each n we can obtain (by means of an algorithm) a certain expression S_n which we think of as stating that $n \in \bar{K}$ in a language we construct. Let us further assume that methods of proof have been provided and that we have an algorithm such that given n and an alleged proof of S_n , we can determine using the algorithm whether or not the alleged proof of S_n really is one. Finally, let us assume that no false statement can be proved using the rules of proof, that is, that

$$\text{if } \vdash S_i, \text{ then } i \in \bar{K}.$$

Here $\vdash S_i$ means that S_i can be proved according to the rules of proof under discussion.

Then we have:

Gödel's Theorem (Kleene-Post Form). There is a number q such that $q \in \bar{K}$ but not $\vdash S_q$.

That is the assertion that $q \in \bar{K}$, is true but unprovable by the given rules of proof.

Proof. Let $M = \{i \mid \vdash S_i\}$. First we note that by what we have assumed above,

$$M \subset \bar{K}. \tag{1}$$

Second, M is re. For, let

$$f(i) = 0, \text{ if } \vdash S_i, \quad f(i) \text{ undefined otherwise. } \text{159}$$

Clearly M is the domain of definition of f . So we need to show that f is partial-recursive. To see that it is, let us first associate integers with each proof, according to the present rules of proof, in a manner similar to what was done above for Turing machines. Now, given an i for which we wish to compute $f(i)$, we generate the integer $0, 1, 2, 3, \dots$ in order and check each one to see whether or not it is the number of a proof of S_i . If it is, we are done, and $f(i) = 0$; otherwise we go on to the next integer.

Since M is re and \bar{K} is not, we have

$$M \neq \bar{K}. \tag{2}$$

Combining (1) and (2) we conclude that some number q belongs to \bar{K} but not to M . This completes the proof.

One should note:

1. The "undecidability" of " $q \in \bar{K}$ " is a relative matter – in the very act of showing it to be undecidable from certain rules of proof, we show that it is true!
2. Using "stronger" rules of proof can only change the value of q .

VI. The Halting Problem for Turing Machines

Let R be a Turing machine which computes a partial-recursive function whose domain of definition is K . Then, we claim that the halting problem for R is unsolvable in the following sense:

There is no algorithm by means of which we may test for given input whether or not R will eventually halt when starting with that input.

For if such an algorithm were available, we could use it to check whether or not a given integer x belongs to K : if and only if R halts eventually when it begins with x written on its tape. Since K is not recursive, no such algorithm can exist.

VII. The Word Problem for Semi-Groups

Let us begin with some alphabet, say a, b, c, d, e , and certain equations between words on this alphabet, e.g.

$$\begin{aligned} bad &= cad \\ ecc &= becd \\ aca &= bd \end{aligned}$$

160 Then certain derived equations will result, e.g.

$$abade = acad = bdde.$$

Generally, given a finite set of equations,

$$\begin{aligned} g_1 &= \overline{g_1} \\ g_2 &= \overline{g_2} \\ &\dots \\ g_K &= \overline{g_K} \end{aligned}$$

the resulting *word problem* is to give an algorithm for determining whether two given words are equal according to the given equations. We shall show how to construct a set of equations for which no such algorithm can exist, i.e. an unsolvable word problem.

We begin with a Turing machine R whose halting problem is unsolvable. Let the internal states of R be q_1, q_2, \dots, q_M and let the symbols it “prints” be S_0, S_1, \dots, S_N , where we take S_0 to be the blank so that “erasing” is taken to mean printing S_0 . We take the tape of R at any given moment to be finite. However motion off the end of the tape is always to be prevented by the addition of a blank square. The entire subsequent history of R ’s computation is determined by specifying at a given moment the corresponding *Post word*:

$$hS_{i(1)}S_{i(2)} \dots S_{i(k)}q_jS_{i(k+1)}S_{i(q)}h$$

where

$$S_{i(1)}S_{i(2)} \dots S_{i(q)}$$

is the word on the tape at this moment of time, q_j is the internal state of R , and S_{k+1} is the symbol scanned. The course of a computation is reflected in a succession of Post words according to the rules (called *semi-Thue productions*):

$$Pq_iS_jQ \rightarrow Pq_lS_kQ \quad (3)$$

whenever $q_iS_jS_kq_l$ is a quadruple of R .

$$\begin{aligned} Pq_iS_jS_kQ &\rightarrow PS_jq_lS_kQ \\ Pq_iS_jhQ &\rightarrow PS_jq_lS_0hQ \end{aligned} \quad (4)$$

whenever $q_i S_j \rightarrow q_l$ is a quadruple of R . ¹⁶¹

$$\begin{aligned} P S_k q_i S_j Q &\rightarrow P q_l S_k S_j Q \\ P h q_i S_j Q &\rightarrow P h q_l S_0 S_j Q \end{aligned} \quad (5)$$

whenever $q_i S_j \rightarrow q_l$, is a quadruple of R .

We introduce two new symbols q and q' and the additional semi-Thue productions:

$$P q_i S_j Q \rightarrow q S_j Q \quad (6)$$

if no quadruple beginning $q_i S_j$ belongs to R , and the productions:

$$P q S_j Q \rightarrow P q Q \quad (7)$$

$$P q h Q \rightarrow P q' h Q \quad (8)$$

$$P S_j q' Q \rightarrow P q' Q \quad (9)$$

The effect of (6) to (9) is to transform the Post word corresponding to the end of a computation (i.e., a time when the machine stops for lack of further instructions) into the word $h q' h$. Hence we have:

R will eventually halt when it begins in a situation described by a given Post word, if and only if the productions (3) to (9) can transform that Post word into $h q' h$.

Now we consider the following equations obtained from (3) to (9):

$$q_i S_j = q_l S_k \quad (3')$$

$$q_i S_j S_k = S_j q_l S_k \quad (4'a)$$

$$q_i S_j h = S_j q_l S_0 h \quad (4'b)$$

$$S_k q_i S_j = q_l S_k S_j \quad (5'a)$$

$$h q_i S_j = h q_l S_0 S_j \quad (5'b)$$

$$q_i S_j = q S_j \quad (6')$$

$$q S_j = q \quad (7')$$

$$q h = q' h \quad (8')$$

$$S_j q' = q' \quad (9')$$

Where the same restrictions are to be observed as in the statements of the corresponding semi-Thue productions.

Then we have: ¹⁶²

Lemma. The equation $P = h q' h$, where P is a Post word, is induced by equations (3') to (9') if and only if Turing machine R eventually halts when beginning in a situation described by Post word P .

Proof. Suppose that R will eventually halt when beginning in a situation described by Post word P . Then, using (3') to (5') we have

$$P = T$$

where T is a Post word corresponding to the end of the computation. Next, using (6') to (9') we have

$$T = h q' h.$$

Next, suppose

$$P = h q' h.$$

Then we have a sequence

$$P_1, P_2, \dots, P_q$$

where P_1 is P and P_q is $h q' h$ and P_{i+1} is obtained directly from P_i using one of the equations (3') to (9').

We wish to show that we can obtain such a sequence in which we use (3') to (9'), but *reading from left to right*, i.e., using just the semi-Thue productions (3) to (9). To see this, let

$$P_{k+1}, P_{k+2}, \dots, P_q$$

be obtained using (3) to (9) (as indeed P_q must be obtained from P_{q-1}), but let P_{k+1} be obtained from P_k using one of the equations (3') to (9') from right to left. But then P_k can be obtained from P_{k+1} using one of equations (3) to (9), i.e., both P_k and P_{k+2} are obtained from P_{k+1} using (3) to (9). But then P_k and P_{k+2} are identical, since no Post

word can have more than one successor under (3) to (9). (This is the “monogenic” character of Turing machines.) Hence, we may eliminate P_k and P_{k+1} from the sequence. Continuing this process, we obtain the desired result.

From the lemma it follows at once that equations (3') to (9') give rise to an unsolvable word problem. For any algorithm for testing the correction of equations could be used to check whether or not

$$P = hq'h$$

for a Post word P and hence to solve the (unsolvable) halting problem for R .

VIII. *Some Concluding Remarks*

In conclusion we should like to indicate various directions which work in the theory of recursive functions has taken. ¹⁶³

1. There has been work on the unsolvability of various specific mathematical problems. Here the unsolvability of the word problem for groups has been the outstanding result.

2. The classification of recursively enumerable sets.

3. How unsolvable is a problem? For example, consider the set

$$U = \{ z \mid \mathcal{A}_z(x) \text{ is everywhere defined} \}.$$

It can be shown that U is not only recursive, but that it is more unsolvable than K in the sense that given an “oracle” which will correctly answer our questions about membership in U , we can give an algorithm for testing for membership in K , but not conversely.

For further details and a bibliography, the reader is referred to the author's *Computability and Unsolvability*, McGraw-Hill, New York, 1958.