Chapter 4

RAM Programs, Turing Machines, and the Partial Recursive Functions

4.1 Introduction

Anyone with some programming experience has an intuituive idea of the notion of "algorithm". Even Euclid's algorithm was called an algorithm long before the invention of modern computers. However, it was not until the 1930's that logicians such as Church, Gödel, Kleene, Turing, and Post, put forth formal definitions for the notions of *effective procedure*, *computable function*, and *algorithm*.

There are surprisingly many different formalizations of the notion of an algorithm. A remarkable fact is that all of these definitions have been shown to be equivalent, in the sense that a function computable in one of these formulations is also computable in all of the others. For most computer scientists, the notion of algorithm is synonymous with that of a program written in some general purpose programming language.

To be more accurate, an algorithm refers to a program that *halts* for all inputs. A program that halts for some inputs but diverges for others is called a *procedure*. One of the characterizations of the computable functions is that they are computed by programs written in a very simple programming language, the language of *RAM programs, also called Post machines*.

Another goal of the theory of computation is to explore the limitations of the computational power of programs. For example, one can ask whether there exists an algorithm that could be used as a debugging tool, to test whether any given program halts on any given input. Another useful program would be one to test whether any two given programs are equivalent for all inputs. As we shall see, such programs do not exist. We have stumbled upon some *undecidable problems*.

Why is a question undecidable, that is, not answerable by a program halting for all inputs? What power must a programming language (or formal system) have, in order that some questions about it are undecidable?

We shall be concerned with these issues as we develop a technical formulation of what is an algorithm.

Before embarking on an extensive study of notions such as algorithms and procedures, a few crucial remarks are in order. Firstly, a program is a *finite* object. It may use a very large amount of memory, but still a *bounded* amount. However, there is no bound on the size of data (strings) held in the registers used by programs. Secondly, all the programming languages under consideration have the property that programs can be effectively *coded* as strings or numbers. This means that there is an algorithm that assigns a code to each program, and conversely, that there is an algorithm that, given a purported code name, tells whether or not the code name represents a program, and if so, which program. For example, we shall see that RAM programs can be encoded as positive natural numbers.

Since we will be dealing with algorithms working on strings or natural numbers, we will have the ability to give as input to a program input data that stand either for a true data, or an encoding for a program. This situation is analogous to that in assembly languages, where a memory word either stands for a data or for an instruction, depending on its interpretation.

It turns out that it is the ability of encoding programs into numbers (or strings) and to decode numbers back into programs, that is often the cause for the undecidability of a question.

In the following Chapters, we study various algorithmic systems. We begin with RAM programs, and continue with Turing machines. It turns out that RAM programs and Turing machine compute precisely the same clas of (partial functions). This famous class of function is called the class of *partial recursive functions*.

Chapter 5

Universal RAM Programs and Undecidability of the Halting Problem

5.1 Pairing Functions

Pairing functions are used to encode pairs of integers into single integers, or more generally, finite sequences of integers into single integers. We begin by exhibiting a bijective pairing function $J\colon \mathbb{N}^2 \to \mathbb{N}$.

The function J has the graph partially showed below:

$$
\begin{array}{ccccccccc}
\vdots & & & & & & & & \\
6 & \ldots & & & & & & & \\
 & \searrow & & & & & & & \\
3 & & 7 & \ldots & & & & & \\
 & \searrow & & \searrow & & & & & \\
1 & & 4 & & 8 & \ldots & & & \\
 & \searrow & & \searrow & & \searrow & & & \\
0 & & 2 & & 5 & & 9 & \ldots &
\end{array}
$$

The function J corresponds to a certain way of enumerating pairs of integers. Note that the value of $x + y$ is constant along each diagonal, and consequently, we have

$$
\begin{aligned}
J(x, y) &= 1 + 2 + \cdots + (x + y) + x, \\
&= ((x + y)(x + y + 1) + 2x)/2, \\
&= ((x + y)^2 + 3x + y)/2,
\end{aligned}
$$

that is,

$$J(x, y) = ((x + y)^2 + 3x + y)/2.$$

Let $K\colon \mathbb{N} \to \mathbb{N}$ and $L\colon \mathbb{N} \to \mathbb{N}$ be the projection functions onto the axes, that is, the unique functions such that

$$K(J(a,b)) = a \quad \text{and} \quad L(J(a,b)) = b,$$

for all $a, b \in \mathbb{N}$. Clearly, J is primitive recursive, since it is given by a polynomial. It is not hard to prove that J is injective and surjective, and that it is strictly monotonic in each argument, which means that for all $x, x', y, y' \in \mathbb{N}$, if $x < x'$ then $J(x,y) < J(x',y)$, and if $y < y'$ then $J(x,y) < J(x,y')$.

The projection functions can be computed explicitly, although this is a bit tricky. We only need to observe that by monotonicity of J,

$$x \leq J(x,y) \quad \text{and} \quad y \leq J(x,y),$$

and thus,

$$K(z) = \min(x \leq z)(\exists y \leq z)[J(x,y) = z],$$

and

$$L(z) = \min(y \leq z)(\exists x \leq z)[J(x,y) = z].$$

The pairing function $J(x,y)$ is also denoted as $\langle x, y\rangle$, and K and L are also denoted as Π_1 and Π_2.

By induction, we can define bijections between $\mathbb{N}^n$ and $\mathbb{N}$ for all $n \geq 1$. We let $\langle z\rangle_1 = z$,

$$\langle x_1, x_2\rangle_2 = \langle x_1, x_2\rangle,$$

and

$$\langle x_1, \ldots, x_n, x_{n+1}\rangle_{n+1} = \langle x_1, \ldots, \langle x_n, x_{n+1}\rangle\rangle_n.$$

Note that

$$\langle x_1, \ldots, x_n, x_{n+1}\rangle_{n+1} = \langle x_1, \langle x_2, \ldots, x_{n+1}\rangle_n\rangle.$$

We can define a uniform projection function Π with the following property:
if $z = \langle x_1, \ldots, x_n\rangle$, with $n \geq 2$, then

$$\Pi(i, n, z) = x_i$$

for all i, where $1 \leq i \leq n$. The idea is to view z as a n-tuple, and $\Pi(i, n, z)$ as the i-th component of that n-tuple. The function Π is defined by cases as follows:

$$\begin{aligned}
\Pi(i,0,z) &= 0, \quad \text{for all } i \geq 0,\\
\Pi(i,1,z) &= z, \quad \text{for all } i \geq 0,\\
\Pi(i,2,z) &= \Pi_1(z), \quad \text{if } 0 \leq i \leq 1,\\
\Pi(i,2,z) &= \Pi_2(z), \quad \text{for all } i \geq 2,
\end{aligned}$$

and for all $n \geq 2$,

$$\Pi(i, n+1, z) = \begin{cases} \Pi(i,n,z) & \text{if } 0 \leq i < n,\\ \Pi_1(\Pi(n,n,z)) & \text{if } i = n,\\ \Pi_2(\Pi(n,n,z)) & \text{if } i > n. \end{cases}$$

By a previous exercise, this is a legitimate primitive recursive definition. Some basic properties of Π are given as exercises. In particular, the following properties are easily shown:

(a) $\langle 0, \ldots, 0\rangle_n = 0$, $\langle x, 0\rangle = \langle x, 0, \ldots, 0\rangle_n$;

(b) $\Pi(0, n, z) = \Pi(1, n, z)$ and $\Pi(i, n, z) = \Pi(n, n, z)$, for all $i \geq n$ and all $n, z \in \mathbb{N}$;

(c) $\langle \Pi(1, n, z), \ldots, \Pi(n, n, z)\rangle_n = z$, for all $n \geq 1$ and all $z \in \mathbb{N}$;

(d) $\Pi(i, n, z) \leq z$, for all $i, n, z \in \mathbb{N}$;

(e) There is a primitive recursive function Large, such that,

$$\Pi(i, n+1, \mathrm{Large}(n+1, z)) = z,$$

for $i, n, z \in \mathbb{N}$.

As a first application, we observe that we need only consider partial recursive functions of a single argument. Indeed, let $\varphi\colon \mathbb{N}^n \to \mathbb{N}$ be a partial recursive function of $n \geq 2$ arguments. Let

$$\overline{\varphi}(z) = \varphi(\Pi(1, n, z), \ldots, \Pi(n, n, z)),$$

for all $z \in \mathbb{N}$. Then, $\overline{\varphi}$ is a partial recursive function of a single argument, and φ can be recovered from $\overline{\varphi}$, since

$$\varphi(x_1, \ldots, x_n) = \overline{\varphi}(\langle x_1, \ldots, x_n\rangle).$$

Thus, using $\langle -, -\rangle$ and Π as coding and decoding functions, we can restrict our attention to functions of a single argument.

Next, we show that there exist coding and decoding functions between Σ^* and $\{a_1\}^*$, and that partial recursive functions over Σ^* can be recoded as partial recursive functions over $\{a_1\}^*$. Since $\{a_1\}^*$ is isomorphic to $\mathbb{N}$, this shows that we can restrict out attention to functions defined over $\mathbb{N}$.

5.2 Equivalence of Alphabets

Given an alphabet $\Sigma = \{a_1, \ldots, a_k\}$, strings over Σ can be ordered by viewing strings as numbers in a number system where the digits are $a_1, \ldots, a_k$. In this number system, which is almost the number system with base k, the string a_1 corresponds to zero, and a_k to $k-1$. Hence, we have a kind of shifted number system in base k. For example, if $\Sigma = \{a, b, c\}$, a listing of Σ^* in the ordering corresponding to the number system begins with

$$a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc,$$
$$aaa, aab, aac, aba, abb, abc, \ldots$$

Clearly, there is an ordering function from Σ^* to $\mathbb{N}$ which is a bijection. Indeed, if $u = a_{i_1} \cdots a_{i_n}$, this function $f\colon \Sigma^* \to \mathbb{N}$ is given by

$$f(u) = i_1 k^{n-1} + i_2 k^{n-2} + \cdots + i_{n-1} k + i_n.$$

Since we also want a decoding function, we define the coding function $C_k\colon \Sigma^* \to \Sigma^*$ as follows:

$C_k(\epsilon) = \epsilon$, and if $u = a_{i_1} \cdots a_{i_n}$, then

$$C_k(u) = a_1^{i_1 k^{n-1} + i_2 k^{n-2} + \cdots + i_{n-1} k + i_n}.$$

The function C_k is primitive recursive, because

$$\begin{aligned} C_k(\epsilon) &= \epsilon, \\ C_k(xa_i) &= C_k(x)^k a_1^i. \end{aligned}$$

The inverse of C_k is a function $D_k\colon \{a_1\}^* \to \Sigma^*$. However, primitive recursive functions are total, and we need to extend D_k to Σ^*. This is easily done by letting

$$D_k(x) = D_k(a_1^{|x|})$$

for all $x \in \Sigma^*$. It remains to define D_k by primitive recursion over $\{a_1\}^*$. For this, we introduce three auxiliary functions p, q, r, defined as follows. Let

$$\begin{aligned} p(\epsilon) &= \epsilon, \\ p(xa_i) &= xa_i, \quad \text{if } i \neq k, \\ p(xa_k) &= p(x). \end{aligned}$$

Note that $p(x)$ is the result of deteting consecutive a_k's in the tail of x. Let

$$\begin{aligned} q(\epsilon) &= \epsilon, \\ q(xa_i) &= q(x)a_1. \end{aligned}$$

Note that $q(x) = a_1^{|x|}$. Finally, let

$$\begin{aligned} r(\epsilon) &= a_1, \\ r(xa_i) &= xa_{i+1}, \quad \text{if } i \neq k, \\ r(xa_k) &= xa_k. \end{aligned}$$

The function r is almost the successor function, for the ordering. Then, the trick is that $D_k(xa_i)$ is the successor of $D_k(x)$ in the ordering, and if

$$D_k(x) = ya_j a_k^n$$

with $j \neq k$, since the successor of $ya_j a_k^n$ is $ya_{j+1} a_k^n$, we can use r. Thus, we have

$$\begin{aligned} D_k(\epsilon) &= \epsilon, \\ D_k(xa_i) &= r(p(D_k(x)))q(D_k(x) - p(D_k(x))). \end{aligned}$$

Then, both C_k and D_k are primitive recursive, and $C \circ D_k = D_k \circ C_k = \text{id}$.

Let $\varphi\colon \Sigma^* \to \Sigma^*$ be a partial function over Σ^*, and let

$$\varphi^+(x_1, \ldots, x_n) = C_k(\varphi(D_k(x_1), \ldots, D_k(x_n))).$$

The function φ^+ is defined over $\{a_1\}^*$. Also, for any partial function ψ over $\{a_1\}^*$, let

$$\psi^\sharp(x_1, \ldots, x_n) = D_k(\psi(C_k(x_1), \ldots, C_k(x_n))).$$

We claim that if ψ is a partial recursive function over $\{a_1\}^*$, then $\psi^\sharp$ is partial recursive over Σ^*, and that if φ is a partial recursive function over Σ^*, then φ^+ is partial recursive over $\{a_1\}^*$.

First, ψ can be extended to Σ^* by letting

$$\psi(x) = \psi(a_1^{|x|})$$

for all $x \in \Sigma^*$, and so, if ψ is partial recursive, then so is $\psi^\sharp$ by composition. This seems equally obvious for φ and φ^+, but there is a difficulty. The problem is that φ^+ is defined as a composition of functions over Σ^*. We have to show how φ^+ can be defined directly over $\{a_1\}^*$ without using any additional alphabet symbols. This is done in Machtey and Young [1], see Section 2.2, Proposition 2.2.3.

Pairing functions can also be used to prove that certain functions are primitive recursive, even though their definition is not a legal primitive recursive definition. For example, consider the *Fibonacci function* defined as follows:

$$\begin{aligned} f(0) &= 1, \\ f(1) &= 2, \\ f(n+2) &= f(n+1) + f(n), \end{aligned}$$

for all $n \in \mathbb{N}$. This is not a legal primitive recursive definition, since $f(n+2)$ depends both on $f(n+1)$ and $f(n)$. In a primitive recursive definition, $g(y+1, \overline{x})$ is only allowed to depend upon $g(y, \overline{x})$.

Definition 5.2.1 Given any function $f\colon \mathbb{N}^n \to \mathbb{N}$, the function $\overline{f}\colon \mathbb{N}^{n+1} \to \mathbb{N}$ defined such that

$$\overline{f}(y, \overline{x}) = \langle f(0, \overline{x}), \ldots, f(y, \overline{x})\rangle_{y+1}$$

is called the *course-of-value function* for f.

The following lemma holds.

Lemma 5.2.2 *Given any function $f\colon \mathbb{N}^n \to \mathbb{N}$, if f is primitive recursive, then so is $\overline{f}$.*

Proof. First, it is necessary to define a function *con* such that if $x = \langle x_1, \ldots, x_m\rangle$ and $y = \langle y_1, \ldots, y_n\rangle$, where $m, n \geq 1$, then

$$con(m, x, y) = \langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle.$$

This fact is left as an exercise. Now, if f is primitive recursive, let

$$\begin{aligned}\overline{f}(0, \overline{x}) &= f(0, \overline{x}),\\ \overline{f}(y+1, \overline{x}) &= con(y+1, \overline{f}(y, \overline{x}), f(y+1, \overline{x})),\end{aligned}$$

showing that $\overline{f}$ is primitive recursive. Conversely, if $\overline{f}$ is primitive recursive, then

$$f(y, \overline{x}) = \Pi(y+1, y+1, \overline{f}(y, \overline{x})),$$

and so, f is primitive recursive. $\square$

Remark: Why is it that

$$\overline{f}(y+1, \overline{x}) = \langle \overline{f}(y, \overline{x}), f(y+1, \overline{x})\rangle$$

does not work?

We define *course-of-value recursion* as follows.

Definition 5.2.3 Given any two functions $g\colon \mathbb{N}^n \to \mathbb{N}$ and $h\colon \mathbb{N}^{n+2} \to \mathbb{N}$, the function $\overline{f}\colon \mathbb{N}^{n+1} \to \mathbb{N}$ is defined by *course-of-value recursion* from g and h if

$$\begin{aligned}f(0, \overline{x}) &= g(\overline{x}),\\ f(y+1, \overline{x}) &= h(y, \overline{f}(y, \overline{x}), \overline{x}).\end{aligned}$$

The following lemma holds.

Lemma 5.2.4 $\overline{f}\colon \mathbb{N}^{n+1} \to \mathbb{N}$ *is defined by course-of-value recursion from g and h and g, h are primitive recursive, then f is primitive recursive.*

Proof. We prove that $\overline{f}$ is primitive recursive. Then, by Lemma 5.2.2, f is also primitive recursive. To prove that $\overline{f}$ is primitive recursive, observe that

$$\begin{aligned}\overline{f}(0, \overline{x}) &= g(\overline{x}),\\ \overline{f}(y+1, \overline{x}) &= con(y+1, \overline{f}(y, \overline{x}), h(y, \overline{f}(y, \overline{x}), \overline{x})).\end{aligned}$$

$\square$

When we use Lemma 5.2.4 to prove that a function is primitive recursive, we rarely bother to construct a formal course-of-value recursion. Indtead, we simply indicate how the value of $f(y+1, \overline{x})$ can be obtained in a primitive recursive manner from $f(0, \overline{x})$ through $f(y, \overline{x})$. Thus, an informal use of Lemma 5.2.4 shows that the Fibonacci function is primitive recursive. A rigorous proof of this fact is left as an exercise.

5.3 Coding of RAM Programs

In this Section, we present a specific encoding of RAM programs which allows us to treat programs as integers. Encoding programs as integers also allows us to have programs that take other programs as input, and we obtain a *universal program*. Universal programs have the property that given two inputs, the first one being the code of a program and the second one an input data, the universal program simulates the actions of the encoded program on the input data. A coding scheme is also called an indexing or a Gödel numbering, in honor to Gödel, who invented this technique.

From results of the previous Chapter, without loss of generality, we can restrict out attention to RAM programs computing partial functions of one argument over $\mathbb{N}$. Furthermore, we only the following kinds of instructions, each instruction being coded as shown below. Since we are considering functions over the natural numbers, which corresponds to a one-letter alphabet, there is only one kind of instruction of the form `add` and `jmp` (and `add` increments by 1 the contents of the specified register Rj).

Ni		`add`	Rj	$code = \langle 1, i, j, 0\rangle$
Ni		`tail`	Rj	$code = \langle 2, i, j, 0\rangle$
Ni		`continue`		$code = \langle 3, i, 1, 0\rangle$
Ni	Rj	`jmp`	Nka	$code = \langle 4, i, j, k\rangle$
Ni	Rj	`jmp`	Nkb	$code = \langle 5, i, j, k\rangle$

Recall that a conditional jump causes a jump to the closest address Nk above or below iff Rj is nonzero, and if Rj is null, the next instruction is executed. We assume that all lines in a RAM program are numbered. This is always feasible, by labeling unnamed instructions with a new and unused line number.

The code of an instruction I is denoted as $\#I$. To simplify the notation, we introduce the following decoding primitive recursive functions Typ, Nam, Reg, and Jmp, defined as follows:

$$\begin{aligned}
\mathrm{Typ}(x) &= \Pi(1, 4, x),\\
\mathrm{Nam}(x) &= \Pi(2, 4, x),\\
\mathrm{Reg}(x) &= \Pi(3, 4, x),\\
\mathrm{Jmp}(x) &= \Pi(4, 4, x).
\end{aligned}$$

The functions yield the type, line number, register name, and line number jumped to, if any, for an instruction coded by x. Note that we have no need to interpret the values of these functions if x does not code an instruction.

We can define the primitive recursive predicate INST, such that $\mathrm{INST}(x)$ holds iff x codes an instruction. First, we need the connective $\supset$ (*implies*), defined such that

$$P \supset Q \quad \text{iff} \quad \neg P \vee Q.$$

Then, $\mathrm{INST}(x)$ holds iff:

$$\begin{gathered}[1 \leq \mathrm{Typ}(x) \leq 5] \wedge [1 \leq \mathrm{Reg}(x)] \wedge \\ [\mathrm{Typ}(x) \leq 3 \supset \mathrm{Jmp}(x) = 0] \wedge \\ [\mathrm{Typ}(x) = 3 \supset \mathrm{Reg}(x) = 1]\end{gathered}$$

Program are coded as follows. If P is a RAM program composed of the n instructions $I_1, \ldots, I_n$, the code of P, denoted as $\#P$, is

$$\#P = \langle n, \#I_1, \ldots, \#I_n \rangle.$$

Recall from a previous exercise that

$$\langle n, \#I_1, \ldots, \#I_n \rangle = \langle n, \langle \#I_1, \ldots, \#I_n \rangle \rangle.$$

We define the primitive recursive functions Ln, Pg, and Line, such that:

$$\begin{aligned}\mathrm{Ln}(x) &= \Pi(1, 2, x), \\ \mathrm{Pg}(x) &= \Pi(2, 2, x), \\ \mathrm{Line}(i, x) &= \Pi(i, \mathrm{Ln}(x), \mathrm{Pg}(x)).\end{aligned}$$

The function Ln yields the length of the program (the number of instructions), Pg yields the sequence of instructions in the program (really, a code for the sequence), and $\mathrm{Line}(i, x)$ yields the code of the ith instruction in the program. Again, if x does not code a program, there is no need to interpret these functions. However, note that by a previous exercise, it happens that

$$\begin{aligned}\mathrm{Line}(0, x) &= \mathrm{Line}(1, x), \quad \text{and} \\ \mathrm{Line}(\mathrm{Ln}(x), x) &= \mathrm{Line}(i, x), \quad \text{for all } i \geq x.\end{aligned}$$

The primitive recursive predicate PROG is defined such that $\mathrm{PROG}(x)$ holds iff x codes a program. Thus, $\mathrm{PROG}(x)$ holds if each line codes an instruction, each jump has an instruction to jump to, and the last instruction is a `continue`. Thus, $\mathrm{PROG}(x)$ holds iff

$$\begin{gathered}\forall i \leq \mathrm{Ln}(x)[i \geq 1 \supset \\ [\mathrm{INST}(\mathrm{Line}(i, x)) \wedge \mathrm{Typ}(\mathrm{Line}(\mathrm{Ln}(x), x)) = 3 \\ \wedge [\mathrm{Typ}(\mathrm{Line}(i, x)) = 4 \supset \\ \exists j \leq i - 1[j \geq 1 \wedge \mathrm{Nam}(\mathrm{Line}(j, x)) = \mathrm{Jmp}(\mathrm{Line}(i, x))]] \wedge \\ [\mathrm{Typ}(\mathrm{Line}(i, x)) = 5 \supset \\ \exists j \leq \mathrm{Ln}(x)[j > i \wedge \mathrm{Nam}(\mathrm{Line}(j, x)) = \mathrm{Jmp}(\mathrm{Line}(i, x))]]]]\end{gathered}$$

Note that we have used the fact proved as an exercise that if f is a primitive recursive function and P is a primitive recursive predicate, then $\exists x \leq f(y) P(x)$ is primitive recursive.

We are now ready to prove a fundamental result in the theory of algorithms. This result points out some of the limitations of the notion of algorithm.

Theorem 5.3.1 *(Undecidability of the halting problem) There is no RAM program P which halts for all inputs and has the following property when started with input x in register $R1$ and with input i in register $R2$ (the other registers being set to zero):*

(1) P halts with output 1 iff i codes a program that eventually halts when started on input x (all other registers set to zero).

(2) P halts with output 0 in $R1$ iff i codes a program that runs forever when started on input x in $R1$ (all other registers set to zero).

(3) If i does not code a program, then P halts with output 2 in $R2$.

Proof. Assume that P is such a RAM program, and let Q be the following program:

	$R2$	$\leftarrow$	$R1$
		P	
$N1$		`continue`	
	$R1$	`jmp`	$N1a$
		`continue`	

The program Q can be translated into a program using only instructions of type 1, 2, 3, 4, 5, described previously, and let q be the code of this program. Let us see what happens if we run this program on input q in $R1$ (all other registers set to zero).

Just after execution of the assignment $R2 \leftarrow R1$, the program P is started with q in both $R1$ and $R2$. Since P is supposed to halt for all inputs, it eventually halts with output 0 or 1 in $R1$. If P halts with output 1 in $R1$, then Q goes into an infinite loop, while if P halts with output 0 in $R1$, then Q halts. But then, because of the definition of P, we see that P says that Q halts when started on input q iff Q loops forever on input q, and that Q loops forever on input q iff Q halts on input q, a contradiction. Therefore, P cannot exist. $\square$

If we identify the notion of algorithm with that of a RAM program which halts for all inputs, the above theorem says that there is no algorithm for deciding whether a RAM program eventually halts for a given input. We say that the halting problem for RAM programs is *undecidable* (or *unsolvable*). The above theorem also implies that the halting problem for Turing machines is undecidable. Indeed, if we had an algorithm for solving the halting problem for Turing machines, we could solve the halting problem for RAM programs as follows: first, apply the algorithm for translating a RAM program into an equivalent Turing machine, and then apply the algorithm solving the halting problem for Turing machines. The argument if typical in computability theory and is called a "reducibility argument".

Our next goal is to define a primitive recursive function that describes the computation of RAM programs. Assume that we have a RAM program P using n registers $R1, \ldots, Rn$, whose contents are denoted as $r_1, \ldots, r_n$. We can code $r_1, \ldots, r_n$ into a single integer $\langle r_1, \ldots, r_n \rangle$. Conversely, every integer x can be viewed as coding the contents of $R1, \ldots, Rn$, by taking the sequence $\Pi(1, n, x), \ldots, \Pi(n, n, x)$. Actually, it is not necessary to know n, the number of registers, if we make the following observation:

$$\mathrm{Reg}(\mathrm{Line}(i, x)) \leq \mathrm{Line}(i, x) \leq \mathrm{Pg}(x)$$

for all $i, x \in \mathbb{N}$. Then, if x codes a program, then $R1, \ldots, Rx$ certainly include all the registers in the program. Also note that from a previous exercise,

$$\langle r_1, \ldots, r_n, 0, \ldots, 0\rangle = \langle r_1, \ldots, r_n, 0\rangle.$$

We now define the primitive recursive functions Nextline, Nextcont, and Comp, describing the computation of RAM programs.

Definition 5.3.2 Let x code a program and let i be such that $1 \leq i \leq \mathrm{Ln}(x)$. The following functions are defined:

(1) $\mathrm{Nextline}(i, x, y)$ is the number of the next instruction to be executed after executing the ith instruction in the program coded by x, where the contents of the registers is coded by y.

(2) $\mathrm{Nextcont}(i, x, y)$ is the code of the contents of the registers after executing the ith instruction in the program coded by x, where the contents of the registers is coded by y.

(3) $\mathrm{Comp}(x, y, m) = \langle i, z\rangle$, where i and z are defined such that after running the program coded by x for m steps, where the initial contents of the program registers are coded by y, the next instruction to be executed has line number i, and z is the code of the current contents of the registers.

Lemma 5.3.3 *The functions* Nextline, Nextcont, *and* Comp, *are primitive recursive.*

Proof. (1) $\mathrm{Nextline}(i, x, y) = i + 1$, unless the ith instruction is a jump and the contents of the register being tested is nonzero:

$\mathrm{Nextline}(i, x, y) =$
$\max j \leq \mathrm{Ln}(x)[j < i \wedge \mathrm{Nam}(\mathrm{Line}(j, x)) = \mathrm{Jmp}(\mathrm{Line}(i, x))]$
if $\mathrm{Typ}(\mathrm{Line}(i, x)) = 4 \wedge \Pi(\mathrm{Reg}(\mathrm{Line}(i, x)), x, y) \neq 0$
$\min j \leq \mathrm{Ln}(x)[j > i \wedge \mathrm{Nam}(\mathrm{Line}(j, x)) = \mathrm{Jmp}(\mathrm{Line}(i, x))]$
if $\mathrm{Typ}(\mathrm{Line}(i, x)) = 5 \wedge \Pi(\mathrm{Reg}(\mathrm{Line}(i, x)), x, y) \neq 0$
$i + 1$ otherwise.

Note that according to this definition, if the ith line is the final `continue`, then Nextline signals that the program has halted by yielding

$$\mathrm{Nextline}(i, x, y) > \mathrm{Ln}(x).$$

(2) We need two auxiliary functions Add and Sub defined as follows.

$\mathrm{Add}(j, x, y)$ is the number coding the contents of the registers used by the program coded by x after register Rj coded by $\Pi(j, x, y)$ has been increased by 1, and

$\mathrm{Sub}(j, x, y)$ codes the contents of the registers after register Rj has been decremented by 1 (y codes the previous contents of the registers). It is easy to see that

$$\mathrm{Sub}(j, x, y) = \min z \leq y[\Pi(j, x, z) = \Pi(j, x, y) - 1$$
$$\wedge \forall k \leq x[0 < k \neq j \supset \Pi(k, x, z) = \Pi(k, x, y)]].$$

The definition of Add is slightly more tricky. We leave as an exercise to the reader to prove that:

$$\mathrm{Add}(j, x, y) = \min z \leq \mathrm{Large}(x, y + 1)$$
$$[\Pi(j, x, z) = \Pi(j, x, y) + 1 \wedge \forall k \leq x[0 < k \neq j \supset \Pi(k, x, z) = \Pi(k, x, y)]],$$

where the function Large is the function defined in an earlier exercise. Then

$$\mathrm{Nextcont}(i, x, y) =$$
$$\mathrm{Add}(\mathrm{Reg}(\mathrm{Line}(i, x), x, y) \quad \text{if} \quad \mathrm{Typ}(\mathrm{Line}(i, x)) = 1$$
$$\mathrm{Sub}(\mathrm{Reg}(\mathrm{Line}(i, x), x, y) \quad \text{if} \quad \mathrm{Typ}(\mathrm{Line}(i, x)) = 2$$
$$y \quad \text{if} \quad \mathrm{Typ}(\mathrm{Line}(i, x)) \geq 3.$$

(3) Recall that $\Pi_1(z) = \Pi(1, 2, z)$ and $\Pi_2(z) = \Pi(2, 2, z)$. The function Comp is defined by primitive recursion as follows:

$$\mathrm{Comp}(x, y, 0) = \langle 1, y \rangle$$
$$\mathrm{Comp}(x, y, m + 1) = \langle \mathrm{Nextline}(\Pi_1(\mathrm{Comp}(x, y, m)), x, \Pi_2(\mathrm{Comp}(x, y, m))),$$
$$\mathrm{Nextcont}(\Pi_1(\mathrm{Comp}(x, y, m)), x, \Pi_2(\mathrm{Comp}(x, y, m))) \rangle.$$

Recall that $\Pi_1(\mathrm{Comp}(x, y, m))$ is the number of the next instruction to be executed and that $\Pi_2(\mathrm{Comp}(x, y, m))$ codes the current contents of the registers. □

We can now reprove that every RAM computable function is partial recursive. Indeed, assume that x codes a program P. We define the partial function End so that for all x, y, where x codes a program and y codes the contents of its registers, $\mathrm{End}(x, y)$ is the number of steps for which the computation runs before halting, if it halts. If the program does not halt, then $\mathrm{End}(x, y)$ is undefined. Since

$$\mathrm{End}(x, y) = \min m[\Pi_1(\mathrm{Comp}(x, y, m)) = \mathrm{Ln}(x)],$$

End is a partial recursive function. However, in general, End is not a total function.

If y is the value of the register $R1$ before the program P coded by x is started, recall that the contents of the registers is coded by $\langle y, 0 \rangle$. Noticing that 0 and 1 do not code programs, we note that if x codes a program, then $x \geq 2$, and $\Pi_1(z) = \Pi(1, x, z)$ is the contents of $R1$ as coded by z. If φ is the partial recursive function computed by the program P coded by x, then we have

$$\varphi(y) = \Pi_1(\Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle)))).$$

Observe that φ is written in the form $\varphi = g \circ \min f$, for some primitive recursive functions f and g.

We can also exhibit a partial recursive function which enumerates all the unary partial recursive functions. It is a *universal function.*

Abusing the notation slightly, we will write $\varphi(x, y)$ for $\varphi(\langle x, y\rangle)$, viewing φ as a function of two arguments (however, φ is really a function of a single argument). We define the function φ_{univ} as follows:

$$\varphi_{univ}(x, y) = \begin{cases} \Pi_1(\Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle)))) & \text{if PROG}(x), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The function φ_{univ} is a partial recursive function with the following property: for every x coding a RAM program P, for every input y,

$$\varphi_{univ}(x, y) = \varphi_x(y),$$

the value of the partial recursive function φ_x computed by the RAM program P coded by x. If x does not code a program, then $\varphi_{univ}(x, y)$ is undefined for all y.

By Lemma **??**, φ_{univ} is not recursive. Indeed, being an enumerating function for the partial recursive functions, it is an enumerating function for the total recursive functions, and thus, it cannot be recursive. Being a partial function saves us from a contradiction.

The existence of the function φ_{univ} leads us to the notion of an *indexing* of the RAM programs. We can define a listing of the RAM programs as follows. If x codes a program (that is, if PROG(x) holds) and P is the program that x codes, we call this program P the xth RAM program and denote it as P_x. If x does not code a program, we let P_x be the program that diverges for every input:

$N1$		`add`	$R1$
$N1$	$R1$	`jmp`	$N1a$
$N1$		`continue`	

Therefore, in all cases, P_x stands for the xth RAM program. Thus, we have a listing of RAM programs, $P_0, P_1, P_2, P_3, \ldots$, such that every RAM program (of the restricted type considered here) appears in the list exactly once, except for the "infinite loop" program. In particular, note that φ_{univ} being a partial recursive function, it is computed by some RAM program UNIV that has a code *univ* and is the program P_{univ} in the list.

Having an indexing of the RAM programs, we also have an indexing of the partial recursive functions.

Definition 5.3.4 For every integer $x \geq 0$, we let P_x be the RAM program coded by x as defined earlier, and φ_x be the partial recursive function computed by P_x.

Remark: Kleene used the notation $\{x\}$ for the partial recursive function coded by x. Due to the potential confusion with singleton sets, we follow Rogers, and use the notation φ_x.

The existence of the universal function φ_{univ} is sufficiently important to be recorded in the following Lemma.

Lemma 5.3.5 *For the indexing of RAM programs defined earlier, there is a universal partial recursive function φ_{univ} such that, for all $x, y \in \mathbb{N}$, if φ_x is the partial recursive function computed by P_x, then*

$$\varphi_x(y) = \varphi_{univ}(\langle x, y\rangle).$$

The program UNIV computing φ_{univ} can be viewed as an *interpreter* for RAM programs. By giving the universal program UNIV the "program" x and the "data" y, we get the result of executing program P_x on input y. We can view the RAM model as a *stored program computer*.

By Theorem 5.3.1 and Lemma 5.3.5, the halting problem for the single program UNIV is undecidable. Otherwise, the halting problem for RAM programs would be decidable, a contradiction. It should be noted that the program UNIV can actually be written (with a certain amount of pain).

The object of the next Section is to show the existence of Kleene's T-predicate. This will yield another important normal form. In addition, the T-predicate is a basic tool in recursion theory.

5.4 Kleene's T-Predicate

In Section 5.3, we have encoded programs. The idea of this Section is to also encode *computations* of RAM programs. Assume that x codes a program, that y is some input (not a code), and that z codes a computation of P_x on input y. The predicate $T(x, y, z)$ is defined as follows:

$T(x, y, z)$ holds iff x codes a RAM program, y is an input, and z codes a halting computation of P_x on input y.

We will show that T is primitive recursive. First, we need to encode computations. We say that z codes a computation of length $n \geq 1$ if

$$z = \langle n + 2, \langle 1, y_0\rangle, \langle i_1, y_1\rangle, \ldots, \langle i_n, y_n\rangle\rangle,$$

where each i_j is the physical location of the next instruction to be executed and each y_j codes the contents of the registers just before execution of the instruction at the location

i_j. Also, y_0 codes the initial contents of the registers, that is, $y_0 = \langle y, 0\rangle$, for some input y. We let $\mathrm{Ln}(z) = \Pi_1(z)$. Note that i_j denotes the physical location of the next instruction to be executed in the sequence of instructions constituting the program coded by x, and not the line number (label) of this instruction. Thus, the first instruction to be executed is in location 1, $1 \leq i_j \leq \mathrm{Ln}(x)$, and $i_{n-1} = \mathrm{Ln}(x)$. Since the last instruction which is executed is the last physical instruction in the program, namely, a `continue`, there is no next instruction to be executed after that, and i_n is irrelevant. Writing the definition of T is a little simpler if we let $i_n = \mathrm{Ln}(x) + 1$.

Definition 5.4.1 The T-predicate is the primitive recursive predicate defined as follows:

$T(x, y, z)$ iff $\mathrm{PROG}(x)$ and $(\mathrm{Ln}(z) \geq 3)$ and
$\forall j \leq \mathrm{Ln}(z) - 3[0 \leq j \supset$
$\mathrm{Nextline}(\Pi_1(\Pi(j+2, \mathrm{Ln}(z), z)), x, \Pi_2(\Pi(j+2, \mathrm{Ln}(z), z))) = \Pi_1(\Pi(j+3, \mathrm{Ln}(z), z))$ and
$\mathrm{Nextcont}(\Pi_1(\Pi(j+2, \mathrm{Ln}(z), z)), x, \Pi_2(\Pi(j+2, \mathrm{Ln}(z), z))) = \Pi_2(\Pi(j+3, \mathrm{Ln}(z), z))$ and
$\Pi_1(\Pi(\mathrm{Ln}(z) - 1, \mathrm{Ln}(z), z)) = \mathrm{Ln}(x)$ and
$\Pi_1(\Pi(2, \mathrm{Ln}(z), z)) = 1$ and
$y = \Pi_1(\Pi_2(\Pi(2, \mathrm{Ln}(z), z)))$ and $\Pi_2(\Pi_2(\Pi(2, \mathrm{Ln}(z), z))) = 0]$

The reader can verify that $T(x, y, z)$ holds iff x codes a RAM program, y is an input, and z codes a halting computation of P_x on input y. In order to extract the output of P_x from z, we define the primitive recursive function Res as follows:

$$\mathrm{Res}(z) = \Pi_1(\Pi_2(\Pi(\mathrm{Ln}(z), \mathrm{Ln}(z), z))).$$

The explanation for this formula is that $\mathrm{Res}(z)$ are the contents of register $R1$ when P_x halts, that is, $\Pi_1(y_{\mathrm{Ln}(z)})$. Using the T-predicate, we get the so-called Kleene normal form.

Theorem 5.4.2 *(Kleene Normal Form) Using the indexing of the partial recursive functions defined earlier, we have*

$$\varphi_x(y) = \mathrm{Res}[\min z(T(x, y, z))],$$

where $T(x, y, z)$ and Res *are primitive recursive.*

Note that the universal function φ_{univ} can be defined as

$$\varphi_{univ}(x, y) = \mathrm{Res}[\min z(T(x, y, z))].$$

There is another important property of the partial recursive functions, namely, that composition is effective. We need two auxiliary primitive recursive functions. The function Conprogs creates the code of the program obtained by concatenating the programs P_x and P_y, and for $i \geq 2$, $\mathrm{Cumclr}(i)$ is the code of the program which clears registers $R2, \ldots, Ri$. To get Cumclr, we can use the function $\mathrm{clr}(i)$ such that $\mathrm{clr}(i)$ is the code of the program

$N1$		`tail`	Ri
$N1$	Ri	`jmp`	$N1a$
N		`continue`	

We leave it as an exercise to prove that clr, Conprogs, and Cumclr, are primitive recursive.

Theorem 5.4.3 *There is a primitive recursive function c such that*

$$\varphi_{c(x,y)} = \varphi_x \circ \varphi_y.$$

Proof. If both x and y code programs, then $\varphi_x \circ \varphi_y$ can be computed as follows: Run P_y, clear all registers but $R1$, then run P_x. Otherwise, let loop be the index of the infinite loop program:

$$c(x,y) = \begin{cases} \text{Conprogs}(y, \text{Conprogs}(\text{Cumclr}(y), x)) & \text{if PROG}(x) \text{ and PROG}(y) \\ \text{loop} & \text{otherwise.} \end{cases}$$

□

Chapter 6

Elementary Recursive Function Theory

6.1 Acceptable Indexings

In Chapter 5, we have exhibited a specific indexing of the partial recursive functions by encoding the RAM programs. Using this indexing, we showed the existence of a universal function φ_{univ} and of a recursive function c, with the property that for all $x, y \in \mathbb{N}$,

$$\varphi_{c(x,y)} = \varphi_x \circ \varphi_y.$$

It is natural to wonder whether the same results hold if a different coding scheme is used or if a different model of computation is used, for example, Turing machines. In other words, we would like to know if our results depend on a specific coding scheme or not. Our previous results showing the characterization of the partial recursive functions being independennt of the specific model used, suggests that it might be possible to pinpoint certain properties of coding schems which would allow an axiomatic development of recursive function theory. What we are aiming at is to find some simple properties of "nice" coding schemes that allow one to proceed without using explicit coding schemes, as long as the above properties hold.

Remarkably, such properties exist. Furthermore, any two coding schemes having these properties are equivalent in a strong sense (effectively equivalent), and so, one can pick any such coding scheme without any risk of losing anything else because the wrong coding scheme was chosen. Such coding schemes, also called indexings, or Gödel numberings, or even programming systems, are called *acceptable indexings*.

Definition 6.1.1 An *indexing* of the partial recursive functions is an infinite sequence $\varphi_0, \varphi_1, \ldots,$ of partial recursive functions that includes all the partial recursive functions of one argument (there might be repetitions, this is why we are not using the term enumeration). An indexing is *universal* if it contains the partial recursive function φ_{univ} such that

$$\varphi_{univ}(i, x) = \varphi_i(x)$$

for all $i, x \in \mathbb{N}$. An indexing is *acceptable* if it is universal and if there is a total recursive function c for composition, such that

$$\varphi_{c(i,j)} = \varphi_i \circ \varphi_j$$

for all $i, j \in \mathbb{N}$.

From Chapter 5, we know that the specific indexing of the partial recursive functions given for RAM programs is acceptable. Another characterization of acceptable indexings left as an exercise is the following: an indexing $\psi_0, \psi_1, \psi_2, \ldots$ of the partial recursive functions is acceptable iff there exists a total recurive function f translating the RAM indexing of Section 5.3 into the indexing $\psi_0, \psi_1, \psi_2, \ldots$, that is,

$$\varphi_i = \psi_{f(i)}$$

for all $i \in \mathbb{N}$.

A very useful property of acceptable indexings is the so-called "s-m-n Theorem". Using the slightly loose notation $\varphi(x_1, \ldots, x_n)$ for $\varphi(\langle x_1, \ldots, x_n \rangle)$, the s-m-n theorem says the following. Given a function φ considered as having $m+n$ arguments, if we fix the values of the first m arguments and we let the other n arguments vary, we obtain a function ψ of n arguments. Then, the index of ψ depends in a recursive fashion upon the index of φ and the first m arguments $x_1, \ldots, x_m$. We can "pull" the first m arguments of φ into the index of ψ.

Theorem 6.1.2 *(The "s-m-n Theorem") For any acceptable indexing $\varphi_0, \varphi_1, \ldots$, there is a total recursive function s, such that, for all $i, m, n \geq 1$, for all $x_1, \ldots, x_m$ and all $y_1, \ldots, y_n$, we have*

$$\varphi_{s(i,m,x_1,\ldots,x_m)}(y_1, \ldots, y_n) = \varphi_i(x_1, \ldots, x_m, y_1, \ldots, y_n).$$

Proof. First, note that the above identity is really

$$\varphi_{s(i,m,\langle x_1,\ldots,x_m \rangle)}(\langle y_1, \ldots, y_n \rangle) = \varphi_i(\langle x_1, \ldots, x_m, y_1, \ldots, y_n \rangle).$$

Recall that there is a primitive recursive function Con such that

$$\mathrm{Con}(m, \langle x_1, \ldots, x_m \rangle, \langle y_1, \ldots, y_n \rangle) = \langle x_1, \ldots, x_m, y_1, \ldots, y_n \rangle$$

for all $x_1, \ldots, x_m, y_1, \ldots, y_n \in \mathbb{N}$. Hence, a recursive function s such that

$$\varphi_{s(i,m,x)}(y) = \varphi_i(\mathrm{Con}(m, x, y))$$

will do. We define some auxiliary primitive recursive functions as follows:

$$P(y) = \langle 0, y \rangle \quad \text{and} \quad Q(\langle x, y \rangle) = \langle x+1, y \rangle.$$

Since we have an indexing of the partial recursive functions, there are indices p and q such that $P = \varphi_p$ and $Q = \varphi_q$. Let R be defined such that

$$R(0) = p,$$
$$R(x+1) = c(q, R(x)),$$

where c is the recursive function for composition given by the indexing. We leave as an exercise to prove that

$$\varphi_{R(x)}(y) = \langle x, y \rangle$$

for all $x, y \in \mathbb{N}$. Also, recall that $\langle x, y, z \rangle = \langle x, \langle y, z \rangle \rangle$, by definition of pairing. Then, we have

$$\varphi_{R(x)} \circ \varphi_{R(y)}(z) = \varphi_{R(x)}(\langle y, z \rangle) = \langle x, y, z \rangle.$$

Finally, let k be an index for the function Con, that is, let

$$\varphi_k(\langle m, x, y \rangle) = \mathrm{Con}(m, x, y).$$

Define s by

$$s(i, m, x) = c(i, c(k, c(R(m), R(x)))).$$

Then, we have

$$\varphi_{s(i,m,x)}(y) = \varphi_i \circ \varphi_k \circ \varphi_{R(m)} \circ \varphi_{R(x)}(y) = \varphi_i(\mathrm{Con}(m, x, y)),$$

as desired. Notice that if the composition function c is primitive recursive, then s is also primitive recursive. In particular, for the specific indexing of the RAM programs given in Section 5.3, the function s is primitive recursive. $\square$

As a first application of the s-m-n Theorem, we show that any two acceptable indexings are effectively inter-translatable.

Theorem 6.1.3 *Let $\varphi_0, \varphi_1, \ldots,$ be a universal indexing, and let $\psi_0, \psi_1, \ldots,$ be any indexing with a total recursive s-1-1 function, that is, a function s such that*

$$\psi_{s(i,1,x)}(y) = \psi_i(x, y)$$

for all $i, x, y \in \mathbb{N}$. Then, there is a total recursive function t such that $\varphi_i = \psi_{t(i)}$.

Proof. Let φ_{univ} be a universal partial recursive function for the indexing $\varphi_0, \varphi_1, \ldots$. Since $\psi_0, \psi_1, \ldots,$ is also an indexing, φ_{univ} occurs somewhere in the second list, and thus, there is some k such that $\varphi_{univ} = \psi_k$. Then, we have

$$\psi_{s(k,1,i)}(x) = \psi_k(i, x) = \varphi_{univ}(i, x) = \varphi_i(x),$$

for all $i, x \in \mathbb{N}$. Therefore, we can take the function t to be the function defined such that

$$t(i) = s(k, 1, i)$$

for all $i \in \mathbb{N}$. $\square$

Using Theorem 6.1.3, if we have two acceptable indexings $\varphi_0, \varphi_1, \ldots$, and $\psi_0, \psi_1, \ldots$, there exist total recursive functions t and u such that

$$\varphi_i = \psi_{t(i)} \quad \text{and} \quad \psi_i = \varphi_{u(i)}$$

for all $i \in \mathbb{N}$. Also note that if the composition function c is primitive recursive, then any s-m-n function is primitive recursive, and the translation functions are primitive recursive. Actually, a stronger result can be shown. It can be shown that for any two acceptable indexings, there exist total recursive *injective* and *surjective* translation functions. In other words, any two acceptable indexings are recursively isomorphic (Roger's isomorphism theorem). Next, we turn to algorithmically unsolvable, or *undecidable*, problems.

6.2 Undecidable Problems

We saw in Section 5.3 that the halting problem for RAM programs is undecidable. In this section, we take a slightly more general approach to study the undecidability of problems, and give some tools for resolving decidability questions.

First, we prove again the undecidability of the halting problem, but this time, for *any* indexing of the partial recursive functions.

Theorem 6.2.1 *(Halting Problem, Abstract Version) Let $\psi_0, \psi_1, \ldots$, be any indexing of the partial recursive functions. Then, the function f defined such that*

$$f(x, y) = \begin{cases} 1 & \textit{if } \psi_x(y) \textit{ is defined,} \\ 0 & \textit{if } \psi_x(y) \textit{ is undefined,} \end{cases}$$

is not recursive.

Proof. Assume that f is recursive, and let g be the function defined such that

$$g(x) = f(x, x)$$

for all $x \in \mathbb{N}$. Then g is also recursive. Let θ be the function defined such that

$$\theta(x) = \begin{cases} 0 & \text{if } g(x) = 0, \\ \text{undefined} & \text{if } g(x) = 1. \end{cases}$$

We claim that θ is not even partial recursive. Observe that θ is such that

$$\theta(x) = \begin{cases} 0 & \text{if } \psi_x(x) \text{ is undefined,} \\ \text{undefined} & \text{if } \psi_x(x) \text{ is defined.} \end{cases}$$

If θ was partial recursive, it would occur in the list as some ψ_i, and we would have

$$\theta(i) = \psi_i(i) = 0 \quad \text{iff} \quad \psi_i(i) \text{ is undefined},$$

a contradiction. Therefore, f and g can't be recursive. $\square$

Observe that the proof of Theorem 6.2.1 does not use the fact that the indexing is universal or acceptable, and thus, the theorem holds for any indexing of the partial recursive functions. The function g defined in the proof of Theorem 6.2.1 is the characteristic function of a set denoted as K, where

$$K = \{x \mid \psi_x(x) \text{ is defined}\}.$$

The set K is an example of a set which is not recursive. Since this fact is quite important, we give the following definition.

Definition 6.2.2 A subset of Σ^* (or a subset of $\mathbb{N}$) is *recursive* iff its characteristic function is a total recursive function.

Using Definition 6.2.2, Theorem 6.2.1 can be restated as follows.

Lemma 6.2.3 *For any indexing $\varphi_0, \varphi_1, \ldots$ of the partial recursive functions (over Σ^* or $\mathbb{N}$), the set $K = \{x \mid \varphi_x(x)$ is defined$\}$ is not recursive.*

Recursive sets allow us to define the concept of a decidable (or undecidable) problem. The idea is to generalize the situation described in Section 5.3 and Section 5.4, where a set of objects, the RAM programs, is encoded into a set of natural numbers, using a coding scheme.

Definition 6.2.4 Let C be a countable set of objects, and let P be a property of objects in C. We view P as the set

$$\{a \in C \mid P(a)\}.$$

A *coding-scheme* is an injective function $\#: C \to \mathbb{N}$ that assigns a unique code to each object in C. The property P is *decidable (relative to #)* iff the set $\{\#(a) \mid a \in C \text{ and } P(a)\}$ is recursive. The property P is *undecidable (relative to #)* iff the set $\{\#(a) \mid a \in C \text{ and } P(a)\}$ is not recursive.

Observe that the decidability of a property P of objects in C depends upon the coding scheme #. Thus, if we are cheating in using a non-effective coding scheme, we may declare that a property is decidabe even though it is not decidable in some reasonable coding scheme. Consequently, we require a coding scheme # to be *effective* in the following sense. Given any object $a \in C$, we can effectively (i.e.. algorithmically) determine its code $\#(a)$. Conversely, given any integer $n \in \mathbb{N}$, we should be able to tell effectively if n is the code of some object in C, and if so, to find this object. In practice, it is always possible to describe the objects in C as strings over some (possibly complex) alphabet Σ (sets of trees, graphs, etc). In such cases, the coding schemes are recursive functions from Σ^* to $\mathbb{N} = \{a_1\}^*$.

For example, let $C = \mathbb{N} \times \mathbb{N}$, where the property P is the equality of the partial functions φ_x and φ_y. We can use the pairing function $\langle -, - \rangle$ as a coding function, and the problem is formally encoded as the recursiveness of the set

$$\{\langle x, y\rangle \mid x, y \in \mathbb{N},\ \varphi_x = \varphi_y\}.$$

In most cases, we don't even bother to describe the coding scheme explicitly, knowing that such a description is routine, although perhaps tedious.

We now show that most properties about programs (except the trivial ones) are undecidable. First, we show that it is undecidable whether a RAM program halts for every input. In other words, it is undecidable whether a procedure is an algorithm. We actually prove a more general fact.

Lemma 6.2.5 *For any acceptable indexing $\varphi_0, \varphi_1, \ldots$ of the partial recursive functions, the set*

$$\mathrm{TOTAL} = \{x \mid \varphi_x \textit{ is a total function}\}$$

is not recursive.

Proof. The proof uses a technique known as reducibility. We try to reduce a set A known to be *nonrecursive* to TOTAL via a recursive function $f\colon A \to \mathrm{TOTAL}$, so that

$$x \in A \quad \text{iff} \quad f(x) \in \mathrm{TOTAL}.$$

If TOTAL were recursive, its characteristic function g would be recursive, and thus, the function $g \circ f$ would be recursive, a contradiction, since A is assumed to be nonrecursive. In the present case, we pick $A = K$. To find the recursive function $f\colon K \to \mathrm{TOTAL}$, we use the s-m-n Theorem. Let θ be the function defined below: for all $x, y \in \mathbb{N}$,

$$\theta(x, y) = \begin{cases} \varphi_x(x) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K. \end{cases}$$

Note that θ does not depend on y. The function θ is partial recursive. Indeed, we have

$$\theta(x, y) = \varphi_x(x) = \varphi_{univ}(x, x).$$

Thus, θ has some index j, so that $\theta = \varphi_j$, and by the s-m-n Theorem, we have

$$\varphi_{s(j,1,x)}(y) = \varphi_j(x, y) = \theta(x, y).$$

Let f be the recursive function defined such that

$$f(x) = s(j, 1, x)$$

for all $x \in \mathbb{N}$. Then, we have

$$\varphi_{f(x)}(y) = \begin{cases} \varphi_x(x) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K \end{cases}$$

for all $y \in \mathbb{N}$. Thus, observe that $\varphi_{f(x)}$ is a total function iff $x \in K$, that is,

$$x \in K \quad \text{iff} \quad f(x) \in \text{TOTAL},$$

where f is recursive. As we explained earlier, this shows that TOTAL is not recursive. $\square$

The above argument can be generalized to yield a result known as Rice's Theorem. Let $\varphi_0, \varphi_1, \ldots$ be any indexing of the partial recursive functions, and let C be any set of partial recursive functions. We define the set P_C as

$$P_C = \{x \in \mathbb{N} \mid \varphi_x \in C\}.$$

We can view C as a property of some of the partial recursive functions. For example

$$C = \{\text{all total recursive functions}\}.$$

We say that C is *nontrivial* if C is neither empty nor the set of all partial recursive functions. Equivalently C is nontrivial iff $P_C \neq \emptyset$ and $P_C \neq \mathbb{N}$. We may think of P_C as the set of programs computing the functions in C.

Theorem 6.2.6 *(Rice's Theorem) For any acceptable indexing $\varphi_0, \varphi_1, \ldots$ of the partial recursive functions, for any set C of partial recursive functions, the set*

$$P_C = \{x \in \mathbb{N} \mid \varphi_x \in C\}$$

is nonrecursive unless C is trivial.

Proof. Assume that C is nontrivial. A set is recursive iff its complement is recursive (the proof is trivial). Hence, we may assume that the totally undefined function is not in C, and since $C \neq \emptyset$, let ψ be some other function in C. We produce a recursive function f such that

$$\varphi_{f(x)}(y) = \begin{cases} \psi(y) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K, \end{cases}$$

for all $y \in \mathbb{N}$. We get f by using the s-m-n Theorem. Let $\psi = \varphi_i$, and define θ as follows:

$$\theta(x, y) = \varphi_{univ}(i, y) + (\varphi_{univ}(x, x) - \varphi_{univ}(x, x)),$$

where $-$ is the primitive recursive function for truncated subtraction. Clearly, θ is partial recursive, and let $\theta = \varphi_j$. By the s-m-n Theorem, we have

$$\varphi_{s(j,1,x)}(y) = \varphi_j(x, y) = \theta(x, y)$$

for all $x, y \in \mathbb{N}$. Letting f be the recursive function such that

$$f(x) = s(j, 1, x),$$

by definition of θ, we get

$$\varphi_{f(x)}(y) = \theta(x, y) = \begin{cases} \psi(y) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K. \end{cases}$$

Thus, f is the desired reduction function. Now, we have

$$x \in K \quad \text{iff} \quad f(x) \in P_C,$$

and thus, the characteristic function C_K of K is equal to $C_P \circ f$, where C_P is the characteristic function of P_C. Therefore, P_C is not recursive, since otherwise, K would be recursive, a contradiction. $\square$

Rice's Theorem shows that all nontrivial properties of the input/output behavior of programs are undecidable! In particular, the following properties are undecidable.

Lemma 6.2.7 *The following properties of partial recursive functions are undecidable.*

(a) A partial recursive function is a constant function.

(b) Given any integer $y \in \mathbb{N}$, is y in the range of some partial recursive function.

(c) Two partial recursive functions φ_x and φ_y are identical.

(d) A partial recursive function φ_x is equal to a given partial recursive function φ_a.

(e) A partial recursive function yields output z on input y, for any given $y, z \in \mathbb{N}$.

(f) A partial recursive function diverges for some input.

(g) A partial recursive function diverges for all input.

The above Lemma is left as an easy exercise.

A property may be undecidable although it is partially decidable. By partially decidable, we mean that there exists a recursive function g that enumerates the set $P_C = \{x \mid \varphi_x \in C\}$. This means that there is a recursive function g whose range is P_C. We say that P_C is *recursively enumerable*. Indeed, g provides a recursive enumeration of P_C, with possible repetitions. Recursively enumerable sets are the object of the next Section.

6.3 Recursively Enumerable Sets

Consider the set

$$A = \{x \in \mathbb{N} \mid \varphi_x(a) \text{ is defined}\},$$

where $a \in \mathbb{N}$ is any fixed natural number. By Rice's Theorem, A is not recursive (check this). We claim that A is the range of a recursive function g. For this, we use the T-predicate. We produce a function which is actually primitive recursive. First, note that A is nonempty (why?), and let x_0 be any index in A. We define g by primitive recursion as follows:

$$g(0) = x_0,$$
$$g(x+1) = \begin{cases} \Pi_1(x) & \text{if } T(\Pi_1(x), a, \Pi_2(x)), \\ x_0 & \text{otherwise.} \end{cases}$$

Since this type of argument is new, it is helpful to explain informally what g does. For every input x, the function g tries finitely many steps of a computation on input a of some partial recursive function. The computation is given by $\Pi_2(x)$, and the partial function is given by $\Pi_1(x)$. Since Π_1 and Π_2 are projection functions, when x ranges over $\mathbb{N}$, both $\Pi_1(x)$ and $\Pi_2(x)$ also range over $\mathbb{N}$. Such a process is called a *dovetailing* computation. Therefore all computations on input a for all partial recursive functions will be tried, and the indices of the partial recursive functions converging on input a will be selected. This type of argument will be used over and over again.

Definition 6.3.1 A subset X of $\mathbb{N}$ is *recursively enumerable* iff either $X = \emptyset$, or X is the range of some total recursive function. Similarly, a subset X of Σ^* is *recursively enumerable* iff either $X = \emptyset$, or X is the range of some total recursive function.

The following Lemma relates recursive sets and recursively enumerable sets.

Lemma 6.3.2 *A set A is recursive iff both A and its complement $\overline{A}$ are recursively enumerable.*

Proof. Assume that A is recursive. Then, it is trivial that its complement is also recursive. Hence, we only have to show that a recursive set is recursively enumerable. The empty set is recursively enumerable by definition. Otherwise, let $y \in A$ be any element. Then, the function f defined such that

$$f(x) = \begin{cases} x & \text{iff } C_A(x) = 1, \\ y & \text{iff } C_A(x) = 0, \end{cases}$$

for all $x \in \mathbb{N}$ is recursive and has range A.

Conversely, assume that both A and $\overline{A}$ are recursively enumerable. If either A or $\overline{A}$ is empty, then A is recursive. Otherwise, let $A = f(\mathbb{N})$ and $\overline{A} = g(\mathbb{N})$, for some recursive functions f and g. We define the function C_A as follows:

$$C_A(x) = \begin{cases} 1 & \text{if } f(\min y[f(y) = x \ \vee \ g(y) = x]) = x, \\ 0 & \text{otherwise.} \end{cases}$$

The function C_A lists A and $\overline{A}$ in parallel, waiting to see whether x turns up in A or in $\overline{A}$. Note that x must eventually turn up either in A or in $\overline{A}$, so that C_A is a total recursive function. □

Our next goal is to show that the recursively enumerable sets can be given several equivalent definitions. We will often abbreviate recursively enumerable as *r.e.*

Lemma 6.3.3 *For any subset A of $\mathbb{N}$, the following properties are equivalent:*

(1) A is empty or A is the range of a primitive recursive function.

(2) A is recursively enumerable.

(3) A is the range of a partial recursive function.

(4) A is the domain of a partial recursive function.

Proof. The implication (1) ⇒ (2) is trivial, since A is r.e. iff either it is empty or it is the range of a (total) recursive function.

To prove the implication (2) ⇒ (3), it suffices to observe that the empty set is the range of the totally undefined function (computed by an infinite loop program), and that a recursive function is a partial recursive function.

The implication (3) ⇒ (4) is shown as follows. Assume that A is the range of φ_i. Define the function f such that

$$f(x) = \min y[T(i, \Pi_1(y), \Pi_2(y)) \ \wedge \ \mathrm{Res}(\Pi_2(y)) = x]$$

for all $x \in \mathbb{N}$. Clearly, f is partial recursive and has domain A.

The implication (4) ⇒ (1) is shown as follows. The only nontrivial case is when A is nonempty. Assume that A is the domain of φ_i. Since $A \neq \emptyset$, the following quantity is defined:

$$\Pi_1(\min y[T(i, \Pi_1(y), \Pi_2(y))]).$$

We define the primitive recursive function f as follows:

$$f(0) = \Pi_1(\min y[T(i, \Pi_1(y), \Pi_2(y))]),$$
$$f(x+1) = \begin{cases} \Pi_1(x) & \text{if } T(i, \Pi_1(x), \Pi_2(x)), \\ f(x) & \text{if } \neg T(i, \Pi_1(x), \Pi_2(x)) \end{cases}.$$

Clearly, A is the range of f. □

More intuitive proofs of the implications (3) ⇒ (4) and (4) ⇒ (1) can be given. Assume that $A \neq \emptyset$ and that $A = range(g)$, where g is a partial recursive function. Assume that g is computed by a RAM program P. To compute $f(x)$, we start computing the sequence

$$g(0), g(1), \ldots$$

looking for x. If x turns up as say $g(n)$, then we output n. Otherwise the computation diverges. Hence, the domain of f is the range of g.

Assume now that A is the domain of some partial recursive function g, and that g is computed by some Turing machine M. Since the case where $A = \emptyset$ is trivial, we may assume that $A \neq \emptyset$, and let $n_0 \in A$ be some chosen element in A. We construct another Turing machine performing the following steps: On input n,

(0) Do one step of the computation of $g(0)$

...

(n) Do $n+1$ steps of the computation of $g(0)$

Do n steps of the computation of $g(1)$

...

Do 2 steps of the computation of $g(n-1)$

Do 1 step of the computation of $g(n)$

During this process, whenever the computation of $g(m)$ halts for some $m \leq n$, we output m. Otherwise, we output n_0.

In this fashion, we will enumerate the domain of g, and since we have constructed a Turing machine that halts for every input, we have a total recursive function.

The following Lemma can easily be shown using the proof technique of Lemma 6.3.3.

Lemma 6.3.4 *(1) There is a recursive function h such that*

$$range(\varphi_x) = dom(\varphi_{h(x)})$$

for all $x \in \mathbb{N}$.

(2) There is a recursive function k such that

$$dom(\varphi_x) = range(\varphi_{k(x)})$$

and $\varphi_{k(x)}$ is total recursive, for all $x \in \mathbb{N}$.

The proof of Lemma 6.3.4 is left as an exercise. Using Lemma 6.3.3, we can prove that K is an r.e. set. Indeed, we have $K = dom(f)$, where

$$f(x) = \varphi_{univ}(x, x)$$

for all $x \in \mathbb{N}$. The set

$$K_0 = \{\langle x, y\rangle \mid \varphi_x(y) \text{ converges}\}$$

is also an r.e. set, since $K_0 = dom(g)$, where

$$g(z) = \varphi_{univ}(\Pi_1(z), \Pi_2(z)),$$

which is partial recursive. The sets K and K_0 are examples of r.e. sets that are not recursive.

We can now prove that there are sets that are not r.e.

Lemma 6.3.5 *For any indexing of the partial recursive functions, the complement $\overline{K}$ of the set*

$$K = \{x \in \mathbb{N} \mid \varphi_x(x) \text{ converges}\}$$

is not recursively enumerable.

Proof. If $\overline{K}$ was recursively enumerable, since K is also recusively enumerable, by Lemma 6.3.2, the set K would be recursive, a contradiction. $\square$

The sets $\overline{K}$ and $\overline{K_0}$ are examples of sets that are not r.e. This shows that the r.e. sets are not closed under complementation. However, we leave it as an exercise to prove that the r.e. sets are closed under union and intersection.

We will prove later on that TOTAL is not r.e. This is rather unpleasant. Indeed, this means that there is no way of effectively listing all algorithms (all total recursive functions). Hence, in a certain sense, the concept of partial recursive function (procedure) is more natural than the concept of a (total) recursive function (algorithm).

The next two Lemmas give other characterizations of the r.e. sets and of the recursive sets. The proofs are left as an exercise.

Lemma 6.3.6 *(1) A set A is r.e. iff either it is finite or it is the range of an injective recursive function.*

(2) A set A is r.e. if either it is empty or it is the range of a monotonic partial recursive function.

(3) A set A is r.e. iff there is a Turing machine M such that, for all $x \in \mathbb{N}$, M halts on x iff $x \in A$.

Lemma 6.3.7 *A set A is recusive iff either it is finite or it is the range of a strictly increasing recursive function.*

Another important result relating the concept of partial recursive function and that of an r.e set is given below.

Theorem 6.3.8 *For every unary partial function f, the following properties are equivalent:*

(1) f is partial recursive.

(2) The set

$$\{\langle x, f(x)\rangle \mid x \in dom(f)\}$$

is r.e.

Proof. Let $g(x) = \langle x, f(x)\rangle$. Clearly, g is partial recursive, and

$$range(g) = \{\langle x, f(x)\rangle \mid x \in dom(f)\}.$$

Conversely, assume that

$$range(g) = \{\langle x, f(x)\rangle \mid x \in dom(f)\}$$

for some recursive function g. Then, we have

$$f(x) = \Pi_2(\min y[\Pi_1(g(y)) = x)])$$

for all $x \in \mathbb{N}$, so that f is partial recursive. $\square$

Using our indexing of the partial recursive functions and Lemma 6.3.3, we obtain an indexing of the r.e sets.

Definition 6.3.9 For any acceptable indexing $\varphi_0, \varphi_1, \ldots$ of the partial recursive functions, we define the enumeration $W_0, W_1, \ldots$ of the r.e. sets by setting

$$W_x = dom(\varphi_x).$$

We now describe a technique for showing that certain sets are r.e but not recursive, or complements of r.e. sets that are not recursive, or not r.e, or neither r.e. nor the complement of an r.e. set. This technique is known as *reducibility*.

6.4 Reducibility and Complete Sets

We already used the notion of reducibility in the proof of Lemma 6.2.5 to show that TOTAL is not recursive.

Definition 6.4.1 A set A is *many-one reducible* to a set B if there is a total recursive function f such that

$$x \in A \quad \text{iff} \quad f(x) \in B$$

for all $x \in A$. We write $A \leq B$, and for short, we say that A is *reducible* to B.

The following simple Lemma is left as an exercise to the reader.

Lemma 6.4.2 *Let A, B, C be subsets of $\mathbb{N}$ (or Σ^*). The following properties hold:*

(1) If $A \leq B$ and $B \leq C$, then $A \leq C$.

(2) If $A \leq B$ then $\overline{A} \leq \overline{B}$.

(3) If $A \leq B$ and B is r.e., then A is r.e.

(4) If $A \leq B$ and A is not r.e., then B is not r.e.

(5) If $A \leq B$ and B is recursive, then A is recursive.

(6) If $A \leq B$ and A is not recursive, then B is not recursive.

Another important concept is the concept of a complete set.

Definition 6.4.3 An r.e. set A is *complete w.r.t. many-one reducibility* iff every r.e. set B is reducible to A, i.e., $B \leq A$.

For simplicity, we will often say *complete* for *complete w.r.t. many-one reducibility*.

Theorem 6.4.4 *The following properties hold:*

(1) If A is complete, B is r.e., and $A \leq B$, then B is complete.

(2) K_0 is complete.

(3) K_0 is reducible to K.

Proof. (1) This is left as a simple exercise.

(2) Let W_x be any r.e. set. Then

$$y \in W_x \quad \text{iff} \quad \langle x, y \rangle \in K_0,$$

and the reduction function is the recursive function f such that

$$f(y) = \langle x, y \rangle$$

for all $y \in \mathbb{N}$.

(3) We use the s-m-n Theorem. First, we leave it as an exercise to prove that there is a recursive function f such that

$$\varphi_{f(x)}(y) = \begin{cases} 1 & \text{if } \varphi_{\Pi_1(x)}(\Pi_2(x)) \text{ converges,} \\ \text{undefined} & \text{otherwise,} \end{cases}$$

for all $x, y \in \mathbb{N}$. Then, for every $z \in \mathbb{N}$,

$$z \in K_0 \quad \text{iff} \quad \varphi_{\Pi_1(z)}(\Pi_2(z)) \text{ converges,}$$

iff $\varphi_{f(z)}(y) = 1$ for all $y \in \mathbb{N}$. However,

$$\varphi_{f(z)}(y) = 1 \quad \text{iff} \quad \varphi_{f(z)}(f(z)) = 1,$$

since $\varphi_{f(z)}$ is a constant function. This means that

$$z \in K_0 \quad \text{iff} \quad f(z) \in K,$$

and f is the desired function. $\square$

As a corollary of Theorem 6.4.4, the set K is also complete.

Definition 6.4.5 Two sets A and B have the same *degree of unsolvability* or are *equivalent* iff $A \leq B$ and $B \leq A$.

Since K and K_0 are both complete, they have the same degree of unsolvability. We will now investigate the reducibility and equivalence of various sets. Recall that

$$\mathrm{TOTAL} = \{x \in \mathbb{N} \mid \varphi_x \text{ is total}\}.$$

We define EMPTY and FINITE, as follows:

$$\mathrm{EMPTY} = \{x \in \mathbb{N} \mid \varphi_x \text{ is undefined for all input}\},$$
$$\mathrm{FINITE} = \{x \in \mathbb{N} \mid \varphi_x \text{ is defined only for finitely many input}\}.$$

Obviously, EMPTY $\subset$ FINITE, and since

$$\mathrm{FINITE} = \{x \in \mathbb{N} \mid \varphi_x \text{ has a finite domain}\},$$

we have

$$\overline{\mathrm{FINITE}} = \{x \in \mathbb{N} \mid \varphi_x \text{ has an infinite domain}\},$$

and thus, TOTAL $\subset \overline{\mathrm{FINITE}}$.

Lemma 6.4.6 *We have* $K_0 \leq \overline{\mathrm{EMPTY}}$.

The proof of Lemma 6.4.6 follows from the proof of Theorem 6.4.4. We also have the following Lemma.

Lemma 6.4.7 *The following properties hold:*

(1) EMPTY *is not r.e.*

(2) $\overline{\mathrm{EMPTY}}$ *is r.e.*

(3) $\overline{K}$ *and* EMPTY *are equivalent.*

(4) $\overline{\mathrm{EMPTY}}$ *is complete.*

Proof. We prove (1) and (3), leaving (2) and (4) as an exercise. First, we show that $\overline{K} \leq$ EMPTY. By the s-m-n Theorem, there exists a recursive function f such that

$$\varphi_{f(x)}(y) = \begin{cases} \varphi_x(x) & \text{if } \varphi_x(x) \text{ converges,} \\ \text{undefined} & \text{if } \varphi_x(x) \text{ diverges,} \end{cases}$$

for all $x, y \in \mathbb{N}$. Note that for all $x \in \mathbb{N}$,

$$x \in \overline{K} \quad \text{iff} \quad f(x) \in \mathrm{EMPTY},$$

and thus, $\overline{K} \leq$ EMPTY. Since $\overline{K}$ is not r.e., EMPTY is not r.e.

By the s-m-n Theorem, there is a recursive function g such that

$$\varphi_{g(x)}(y) = \min z[T(x, \Pi_1(z), \Pi_2(z))],$$

for all $x, y \in \mathbb{N}$. Note that

$$x \in \text{EMPTY} \quad \text{iff} \quad g(x) \in \overline{K}$$

for all $x \in \mathbb{N}$. Therefore, EMPTY $\leq \overline{K}$, and since we just showed that $\overline{K} \leq$ EMPTY, the sets $\overline{K}$ and EMPTY are equivalent. $\square$

Lemma 6.4.8 *The following properties hold:*

(1) TOTAL *and* $\overline{\text{TOTAL}}$ *are not r.e.*

(2) FINITE *and* $\overline{\text{FINITE}}$ *are not r.e.*

Proof. Checking the proof of Theorem 6.4.4, we note that $K_0 \leq$ TOTAL and $K_0 \leq \overline{\text{FINITE}}$. Hence, we get $\overline{K_0} \leq \overline{\text{TOTAL}}$ and $\overline{K_0} \leq$ FINITE, and neither $\overline{\text{TOTAL}}$ not FINITE is r.e. If TOTAL was r.e., then there would be a recursive function f such that TOTAL $= range(f)$. Define g as follows:

$$g(x) = \varphi_{f(x)}(x) + 1 = \varphi_{univ}(f(x), x) + 1$$

for all $x \in \mathbb{N}$. Since f is total and $\varphi_{f(x)}$ is total for all $x \in \mathbb{N}$, the function g is total recursive. Let e be an index such that

$$g = \varphi_{f(e)}.$$

Since g is total, $g(e)$ is defined. Then, we have

$$g(e) = \varphi_{f(e)}(e) + 1 = g(e) + 1,$$

a contradiction. Hence, TOTAL is not r.e. Finally, we show that TOTAL $\leq \overline{\text{FINITE}}$. This also shows that $\overline{\text{FINITE}}$ is not r.e. By the s-m-n Theorem, there is a recursive function f such that

$$\varphi_{f(x)}(y) = \begin{cases} 1 & \text{if } \forall z \leq y(\varphi_x(z)\downarrow), \\ \text{undefined} & \text{otherwise,} \end{cases}$$

for all $x, y \in \mathbb{N}$. It is easily seen that

$$x \in \text{TOTAL} \quad \text{iff} \quad f(x) \in \overline{\text{FINITE}}$$

for all $x \in \mathbb{N}$. $\square$

From Lemma 6.4.8, we have TOTAL $\leq \overline{\text{FINITE}}$. It turns out that $\overline{\text{FINITE}} \leq$ TOTAL, and TOTAL and $\overline{\text{FINITE}}$ are equivalent.

Lemma 6.4.9 *The sets* TOTAL *and* $\overline{\text{FINITE}}$ *are equivalent.*

Proof. We show that $\overline{\text{FINITE}} \leq \text{TOTAL}$. By the s-m-n Theorem, there is a recursive function f such that

$$\varphi_{f(x)}(y) = \begin{cases} 1 & \text{if } \exists z \geq y(\varphi_x(z)\downarrow), \\ \text{undefined} & \text{if } \forall z \geq y(\varphi_x(z)\uparrow), \end{cases}$$

for all $x, y \in \mathbb{N}$. It is easily seen that

$$x \in \overline{\text{FINITE}} \quad \text{iff} \quad f(x) \in \text{TOTAL}$$

for all $x \in \mathbb{N}$. $\square$

We now turn to the recursion Theorem.

6.5 The Recursion Theorem

The recursion Theorem, due to Kleene, is a fundamental result in recursion theory. Let f be a total recursive function. Then, it turns out that there is some n such that

$$\varphi_n = \varphi_{f(n)}.$$

Theorem 6.5.1 *(Recursion Theorem, Version 1) Let $\varphi_0, \varphi_1, \ldots$ be any acceptable indexing of the partial recursive functions. For every total recursive function f, there is some n such that*

$$\varphi_n = \varphi_{f(n)}.$$

Proof. Consider the function θ defined such that

$$\theta(x, y) = \varphi_{univ}(\varphi_{univ}(x, x), y)$$

for all $x, y \in \mathbb{N}$. The function θ is partial recursive, and there is some index j such that $\varphi_j = \theta$. By the s-m-n Theorem, there is a recursive function g such that

$$\varphi_{g(x)}(y) = \theta(x, y).$$

Consider the function $f \circ g$. Since it is recursive, there is some index m such that $\varphi_m = f \circ g$. Let

$$n = g(m).$$

Since φ_m is total, $\varphi_m(m)$ is defined, and we have

$$\begin{aligned} \varphi_n(y) = \varphi_{g(m)}(y) = \theta(m, y) &= \varphi_{univ}(\varphi_{univ}(m, m), y) = \varphi_{\varphi_{univ}(m,m)}(y) \\ &= \varphi_{\varphi_m(m)}(y) = \varphi_{f \circ g(m)}(y) = \varphi_{f(g(m))}(y) = \varphi_{f(n)}(y), \end{aligned}$$

for all $y \in \mathbb{N}$. Therefore, $\varphi_n = \varphi_{f(n)}$, as desired. $\square$

The recursion Theorem can be strengthened as follows.

Theorem 6.5.2 *(Recursion Theorem, Version 2) Let $\varphi_0, \varphi_1, \ldots$ be any acceptable indexing of the partial recursive functions. There is a total recursive function h such that for all $x \in \mathbb{N}$, if φ_x is total, then*

$$\varphi_{\varphi_x(h(x))} = \varphi_{h(x)}.$$

Proof. The recursive function g obtained in the proof of Theorem 6.5.1 satisfies the condition

$$\varphi_{g(x)} = \varphi_{\varphi_x(x)},$$

and it has some index i such that $\varphi_i = g$. Recall that c is a recursive composition function such that

$$\varphi_{c(x,y)} = \varphi_x \circ \varphi_y.$$

It is easily verified that the function h defined such that

$$h(x) = g(c(x, i))$$

for all $x \in \mathbb{N}$ does the job. □

A third version of the recursion Theorem is given below.

Theorem 6.5.3 *(Recursion Theorem, Version 3) For all $n \geq 1$, there is a total recursive function h of $n + 1$ arguments, such that for all $x \in \mathbb{N}$, if φ_x is a total recursive function of $n + 1$ arguments, then*

$$\varphi_{\varphi_x(h(x,x_1,\ldots,x_n),x_1,\ldots,x_n)} = \varphi_{h(x,x_1,\ldots,x_n)},$$

for all $x_1, \ldots, x_n \in \mathbb{N}$.

Proof. Let θ be the function defined such that

$$\theta(x, x_1, \ldots, x_n, y) = \varphi_{\varphi_x(x,x_1,\ldots,x_n)}(y) = \varphi_{univ}(\varphi_{univ}(x, x, x_1, \ldots, x_n), y)$$

for all $x, x_1, \ldots, x_n, y \in \mathbb{N}$. By the s-m-n Theorem, there is a recursive function g such that

$$\varphi_{g(x,x_1,\ldots,x_n)} = \varphi_{\varphi_x(x,x_1,\ldots,x_n)}.$$

It is easily shown that there is a recursive function c such that

$$\varphi_{c(i,j)}(x, x_1, \ldots, x_n) = \varphi_i(\varphi_j(x, x_1, \ldots, x_n), x_1, \ldots, x_n)$$

for any two partial recursive functions φ_i and φ_j (viewed as functions of $n + 1$ arguments) and all $x, x_1, \ldots, x_n \in \mathbb{N}$. Let $\varphi_i = g$, and define h such that

$$h(x, x_1, \ldots, x_n) = g(c(x, i), x_1, \ldots, x_n),$$

for all $x, x_1, \ldots, x_n \in \mathbb{N}$. We have

$$\varphi_{h(x,x_1,\ldots,x_n)} = \varphi_{g(c(x,i),x_1,\ldots,x_n)} = \varphi_{\varphi_{c(x,i)}(c(x,i),x_1,\ldots,x_n)},$$

and

$$\begin{aligned}\varphi_{\varphi_{c(x,i)}(c(x,i),x_1,\ldots,x_n)} &= \varphi_{\varphi_x(\varphi_i(c(x,i),x_1,\ldots,x_n),x_1,\ldots,x_n)},\\ &= \varphi_{\varphi_x(g(c(x,i),x_1,\ldots,x_n),x_1,\ldots,x_n)},\\ &= \varphi_{\varphi_x(h(x,x_1,\ldots,x_n),x_1,\ldots,x_n)}.\end{aligned}$$

□

As a first application of the recursion theorem, we can show that there is an index n such that φ_n is the constant function with output n. Loosely speaking, φ_n prints its own name. Let f be the recursive function such that

$$f(x, y) = x$$

for all $x, y \in \mathbb{N}$. By the s-m-n Theorem, there is a recursive function g such that

$$\varphi_{g(x)}(y) = f(x, y) = x$$

for all $x, y \in \mathbb{N}$. By the recursion Theorem 6.5.1, there is some n such that

$$\varphi_{g(n)} = \varphi_n,$$

the constant function with value n.

As a second application, we get a very short proof of Rice's Theorem. Let C be such that $P_C \neq \emptyset$ and $P_C \neq \mathbb{N}$, and let $j \in P_C$ and $k \in \mathbb{N} - P_C$. Define the function f as follows:

$$f(x) = \begin{cases} j & \text{if } x \notin P_C, \\ k & \text{if } x \in P_C, \end{cases}$$

If P_C is recursive, then f is recursive. By the recursion Theorem 6.5.1, there is some n such that

$$\varphi_{f(n)} = \varphi_n.$$

But then, we have

$$n \in P_C \quad \text{iff} \quad f(n) \notin P_C$$

by definition of f, and thus,

$$\varphi_{f(n)} \neq \varphi_n,$$

a contradiction. Hence, P_C is not recursive.

As a third application, we prove the following Lemma.

Lemma 6.5.4 *Let C be a set of partial recursive functions and let*

$$A = \{x \in \mathbb{N} \mid \varphi_x \in C\}.$$

The set A is not reducible to its complement $\overline{A}$.

Proof. Assume that $A \leq \overline{A}$. Then, there is a recursive function f such that

$$x \in A \quad \text{iff} \quad f(x) \in \overline{A}$$

for all $x \in \mathbb{N}$. By the recursion Theorem, there is some n such that

$$\varphi_{f(n)} = \varphi_n.$$

But then,

$$\varphi_n \in C \quad \text{iff} \quad n \in A \quad \text{iff} \quad f(n) \in \overline{A} \quad \text{iff} \quad \varphi_{f(n)} \in \overline{C},$$

contradicting the fact that

$$\varphi_{f(n)} = \varphi_n.$$

□

The recursion Theorem can also be used to show that functions defined by recursive definitions other than primitive recursion are partial recursive. This is the case for the function known as *Ackermann's function*, defined recursively as follows:

$$\begin{aligned} f(0, y) &= y + 1, \\ f(x + 1, 0) &= f(x, 1), \\ f(x + 1, y + 1) &= f(x, f(x + 1, y)). \end{aligned}$$

It can be shown that this function is not primitive recursive. Intuitively, it outgrows all primitive recursive functions. However, f is recursive, but this is not so obvious. We can use the recursion Theorem to prove that f is recursive. Consider the following definition by cases:

$$\begin{aligned} g(n, 0, y) &= y + 1, \\ g(n, x + 1, 0) &= \varphi_{univ}(n, x, 1), \\ g(n, x + 1, y + 1) &= \varphi_{univ}(n, x, \varphi_{univ}(n, x + 1, y)). \end{aligned}$$

Clearly, g is partial recursive. By the s-m-n Theorem, there is a recursive function h such that

$$\varphi_{h(n)}(x, y) = g(n, x, y).$$

By the recursion Theorem, there is an m such that

$$\varphi_{h(m)} = \varphi_m.$$

Therefore, the partial recursive function $\varphi_m(x, y)$ satisfies the definition of Ackermann's function. We showed in a previous Section that $\varphi_m(x, y)$ is a total function, and thus, Ackermann's function is a total recursive function.

Hence, the recursion Theorem justifies the use of certain recursive definitions. However, note that there are some recursive definitions that are only satisfied by the completely undefined function.

In the next Section, we prove the extended Rice Theorem.

6.6 Extended Rice Theorem

The extended Rice Theorem characterizes the sets of partial recursive functions C such that P_C is r.e. First, we need to discuss a way of indexing the partial recursive functions that have a finite domain. Using the uniform projection function Π, we define the primitive recursive function F such that

$$F(x, y) = \Pi(y + 1, \Pi_1(x) + 1, \Pi_2(x)).$$

We also define the sequence of partial functions $P_0, P_1, \ldots$ as follows:

$$P_x(y) = \begin{cases} F(x, y) - 1 & \text{if } 0 < F(x, y) \text{ and } y < \Pi_1(x) + 1, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Lemma 6.6.1 *Every P_x is a partial recursive function with finite domain, and every partial recursive function with finite domain is equal to some P_x.*

The proof is left as an exercise. The easy part of the extended Rice Theorem is the following lemma. Recall that given any two partial functions $f\colon A \to B$ and $g\colon A \to B$, we say that g *extends* f iff $f \subseteq g$, which means that $g(x)$ is defined whenever $f(x)$ is defined, and if so, $g(x) = f(x)$.

Lemma 6.6.2 *Let C be a set of partial recursive functions. If there is an r.e. set A such that, $\varphi_x \in C$ iff there is some $y \in A$ such that φ_x extends P_y, then $P_C = \{x \mid \varphi_x \in C\}$ is r.e.*

Proof. Lemma 6.6.2 can be restated as

$$P_C = \{x \mid \exists y \in A,\ P_y \subseteq \varphi_x\}$$

is r.e. If A is empty, so is P_C, and P_C is r.e. Otherwise, let f be a recursive function such that

$$A = range(f).$$

Let ψ be the following partial recursive function:

$$\psi(z) = \begin{cases} \Pi_1(z) & \text{if } P_{f(\Pi_2(z))} \subseteq \varphi_{\Pi_1(z)}, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

It is clear that

$$P_C = range(\psi).$$

To see that ψ is partial recursive, write $\psi(z)$ as follows:

$$\psi(z) = \begin{cases} \Pi_1(z) & \text{if } \forall w \leq \Pi_1(f(\Pi_2(z)))[F(f(\Pi_2(z)), w) > 0 \\ & \qquad \supset \varphi_{\Pi_1(z)}(w) = F(f(\Pi_2(z)), w) - 1], \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$\square$

To establish the converse of Lemma 6.6.2, we need two Lemmas.

Lemma 6.6.3 *If P_C is r.e. and $\varphi \in C$, then there is some $P_y \subseteq \varphi$ such that $P_y \in C$.*

Proof. Assume that P_C is r.e. and that $\varphi \in C$. By an s-m-n construction, there is a recursive function g such that

$$\varphi_{g(x)}(y) = \begin{cases} \varphi(y) & \text{if } \forall z \leq y[\neg T(x,x,z)], \\ \text{undefined} & \text{if } \exists z \leq y[T(x,x,z)], \end{cases}$$

for all $x, y \in \mathbb{N}$. Observe that if $x \in K$, then $\varphi_{g(x)}$ is a finite subfunction of φ, and if $x \in \overline{K}$, then $\varphi_{g(x)} = \varphi$. Assume that no finite subfunction of φ is in C. Then,

$$x \in \overline{K} \quad \text{iff} \quad g(x) \in P_C$$

for all $x \in \mathbb{N}$, that is, $\overline{K} \leq P_C$. Since P_C is r.e., $\overline{K}$ would also be r.e., a contradiction. $\square$

As a corollary of Lemma 6.6.3, we note that TOTAL is not r.e.

Lemma 6.6.4 *If P_C is r.e., $\varphi \in C$, and $\varphi \subseteq \psi$, where ψ is a partial recursive function, then $\psi \in C$.*

Proof. Assume that P_C is r.e. We claim that there is a recursive function h such that

$$\varphi_{h(x)}(y) = \begin{cases} \psi(y) & \text{if } x \in K, \\ \varphi(y) & \text{if } x \in \overline{K}, \end{cases}$$

for all $x, y \in \mathbb{N}$. Assume that $\psi \notin C$. Then

$$x \in \overline{K} \quad \text{iff} \quad h(x) \in P_C$$

for all $x \in \mathbb{N}$, that is, $\overline{K} \leq P_C$, a contradiction, since P_C is r.e. Therefore, $\psi \in C$. The function h is defined as follows. Let $\varphi = \varphi_j$. Define θ such that

$$\theta(x,y) = \text{if} \quad T(j,y,m_0) \ \wedge \neg T(x,x,m_0) \text{ then} \quad \varphi(y) \quad \text{else} \quad \psi(y),$$

for all $x, y \in \mathbb{N}$, where

$$m_0 = \min m[T(j,y,m) \vee T(x,x,m)].$$

By the s-m-n Theorem, there is a recursive function h such that

$$\varphi_{h(x)}(y) = \theta(x,y)$$

for all $x, y \in \mathbb{N}$, and it is easily seen that θ is the desired function. $\square$

Observe that Lemma 6.6.4 yields a new proof that $\overline{\text{TOTAL}}$ is not r.e. Finally, we can prove the extended Rice Theorem.

Theorem 6.6.5 *(Extended Rice Theorem) The set P_C is r.e. iff there is an r.e. set A such that*

$$\varphi_x \in C \quad \textit{iff} \quad \exists y \in A\, (P_y \subseteq \varphi_x).$$

Proof. Let $P_C = dom(\varphi_i)$. Using the s-m-n Theorem, there is a recursive function k such that

$$\varphi_{k(y)} = P_y$$

for all $y \in \mathbb{N}$. Define the r.e. set A such that

$$A = dom(\varphi_i \circ k).$$

Then,

$$y \in A \quad \text{iff} \quad \varphi_i(k(y)) \downarrow \quad \text{iff} \quad P_y \in C.$$

Next, using Lemma 6.6.3 and Lemma 6.6.4, it is easy to see that

$$\varphi_x \in C \quad \text{iff} \quad \exists y \in A\, (P_y \subseteq \varphi_x).$$

Indeed, if $\varphi_x \in C$, by Lemma 6.6.3, there is a finite subfunction $P_y \subseteq \varphi_x$ such that $P_y \in C$, but

$$P_y \in C \quad \text{iff} \quad y \in A,$$

as desired. On the other hand, if

$$P_y \subseteq \varphi_x$$

for some $y \in A$, then

$$P_y \in C,$$

and by Lemma 6.6.4, since φ_x extends P_y, we get

$$\varphi_x \in C.$$

□

6.7 Creative and Productive Sets

In this section, we discuss some special sets that have important applications in logic: creative and productive sets. The concepts to be described are illustrated by the following situation. Assume that

$$W_x \subseteq \overline{K}$$

for some $x \in \mathbb{N}$. We claim that

$$x \in \overline{K} - W_x.$$

Indeed, if $x \in W_x$, then $\varphi_x(x)$ is defined, and by definition of K, we get $x \notin \overline{K}$, a contradiction. Therefore, $\varphi_x(x)$ must be undefined, that is,

$$x \in \overline{K} - W_x.$$

The above situation can be generalized as follows.

Definition 6.7.1 A set A is *productive* iff there is a total recursive function f such that

$$\text{if} \quad W_x \subseteq A \quad \text{then} \quad f(x) \in A - W_x$$

for all $x \in \mathbb{N}$. The function f is called the *productive function of* A. A set A is *creative* if it is r.e. and if its complement $\overline{A}$ is productive.

As we just showed, K is creative and $\overline{K}$ is productive. The following facts are immediate conequences of the definition.

(1) A productive set is not r.e.

(2) A creative set is not recursive.

Creative and productive sets arise in logic. The set of theorems of a logical theory is often creative. For example, the set of theorems in Peano's arithmetic is creative. This yields incompleteness results.

Lemma 6.7.2 *If a set A is productive, then it has an infinite r.e. subset.*

Proof. We first give an informal proof. let f be the recursive productive function of A. We define a recursive function g as follows: Let x_0 be an index for the empty set, and let

$$g(0) = f(x_0).$$

Assuming that

$$\{g(0), g(1), \ldots, g(y)\}$$

is known, let x_{y+1} be an index for this finite set, and let

$$g(y+1) = f(x_{y+1}).$$

Since $W_{x_{y+1}} \subseteq A$, we have $f(x_{y+1}) \in A$.

For the formal proof, we use the following facts whose proof is left as an exercise:

(1) There is a recursive function u such that

$$W_{u(x,y)} = W_x \cup W_y.$$

(2) There is a recursive function t such that

$$W_{t(x)} = \{x\}.$$

Letting x_0 be an index for the empty set, we define the function h as follows:

$$\begin{aligned} h(0) &= x_0, \\ h(y+1) &= u(t(f(y)), h(y)). \end{aligned}$$

We define g such that

$$g = f \circ h.$$

It is easily seen that g does the job. $\square$

Another important property of productive sets is the following.

Lemma 6.7.3 *If a set A is productive, then $\overline{K} \leq A$.*

Proof. Let f be a productive function for A. Using the s-m-n Theorem, we can find a recursive function h such that

$$W_{h(y,x)} = \begin{cases} \{f(y)\} & \text{if } x \in K, \\ \emptyset & \text{if } x \in \overline{K}. \end{cases}$$

The above can be restated as follows:

$$\varphi_{h(y,x)}(z) = \begin{cases} 1 & \text{if } x \in K \text{ and } z = f(y), \\ \text{undefined} & \text{if } x \in \overline{K}, \end{cases}$$

for all $x, y, z \in \mathbb{N}$. By the third version of the recursion Theorem (Theorem 6.5.3), there is a recursive function g such that

$$W_{g(x)} = W_{h(g(x),x)}$$

for all $x \in \mathbb{N}$. Let

$$k = f \circ g.$$

We claim that

$$x \in \overline{K} \quad \text{iff} \quad k(x) \in A$$

for all $x \in \mathbb{N}$. The verification of this fact is left as an exercise. Thus, $\overline{K} \leq A$. $\square$

Using Lemma 6.7.3, the following results can be shown.

Lemma 6.7.4 *The following facts hold.*

(1) If A is productive and $A \leq B$, then B is productive.

(2) A is creative iff A is equivalent to K.

(3) A is creative iff A is complete,

Bibliography

[1] Michael Machtey and Paul Young. *An Introduction to the General Theory of Algorithms.* Elsevier North-Holland, first edition, 1978.