Chapter 6

Universal RAM Programs and Undecidability of the Halting Problem

6.1 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:

$$\begin{tikzcd}
\vdots & & & \\
6 \ \ldots \arrow[dr] & & & \\
3 \arrow[dr] & 7 \ \ldots \arrow[dr] & & \\
1 \arrow[dr] & 4 \arrow[dr] & 8 \ \ldots \arrow[dr] & \\
0 & 2 & 5 & 9 \ \ldots
\end{tikzcd}$$

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 a recursive function (even primitive recursive), since it is given by a polynomial.

It can be shown 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 K and L 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

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

These functions can be computed by RAM programs involving two nested loops. Thus, they are recursive (in fact, primitive recursive).

More explicit formulae can be given for K and L.

If we define

$$\begin{aligned} Q_1(z) &= \lfloor (\lfloor \sqrt{8z+1} \rfloor + 1)/2 \rfloor - 1 \\ Q_2(z) &= 2z - (Q_1(z))^2, \end{aligned}$$

then it can be shown that

$$\begin{aligned} K(z) &= \frac{1}{2}(Q_2(z) - Q_1(z)) \\ L(z) &= Q_1(z) - \frac{1}{2}(Q_2(z) - Q_1(z)). \end{aligned}$$

In the above formula, the function $m \mapsto \lfloor \sqrt{m} \rfloor$ yields the largest integer s such that $s^2 \leq m$. It can be computed by a RAM program.

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

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

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 function Π is defined by cases as follows:

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

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.

It can be shown 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 $\mathbb{N}$.

6.2 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 need the following kinds of instructions, each instruction being coded as shown below. Because we are only considering functions over $\mathbb{N}$, 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).

$$\begin{array}{lllll}
Ni & & \texttt{add} & Rj & \textit{code} = \langle 1, i, j, 0\rangle \\
Ni & & \texttt{tail} & Rj & \textit{code} = \langle 2, i, j, 0\rangle \\
Ni & & \texttt{continue} & & \textit{code} = \langle 3, i, 1, 0\rangle \\
Ni & Rj & \texttt{jmp} & Nka & \textit{code} = \langle 4, i, j, k\rangle \\
Ni & Rj & \texttt{jmp} & Nkb & \textit{code} = \langle 5, i, j, k\rangle
\end{array}$$

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.

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, INST(x) holds iff:

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

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

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

Also recall that

$$\langle x, y\rangle = ((x+y)^2 + 3x + y)/2.$$

Consider the following program Padd2 computing the function add2: $\mathbb{N} \to \mathbb{N}$ given by

$$\mathrm{add2}(n) = n + 2.$$

```
I1 :   1        add     R1
I2 :   2        add     R1
I3 :   3     continue
```

We have

$$\begin{aligned}
\#I1 &= \langle 1, 1, 1, 0\rangle_4 = \langle 1, \langle 1, \langle 1, 0\rangle\rangle\rangle = 37\\
\#I2 &= \langle 1, 2, 1, 0\rangle_4 = \langle 1, \langle 2, \langle 1, 0\rangle\rangle\rangle = 92\\
\#I3 &= \langle 3, 3, 1, 0\rangle_4 = \langle 3, \langle 3, \langle 1, 0\rangle\rangle\rangle = 234
\end{aligned}$$

and

$$\begin{aligned}\#\text{Padd2} &= \langle 3, \#I1, \#I2, \#I3\rangle_4 = \langle 3, \langle 37, \langle 92, 234\rangle\rangle \\ &= 1\,018\,748\,519\,973\,070\,618.\end{aligned}$$

The codes get big fast!

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

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

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 Line(i, x) yields the code of the ith instruction in the program.

If x does not code a program, there is no need to interpret these functions.

The (primitive) recursive predicate PROG is defined such that PROG(x) holds iff x codes a program.

Thus, 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, PROG(x) holds iff

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

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 6.1. *(Undecidability of the halting problem) There is no RAM program* **Decider** *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) **Decider** *halts with output 1 iff i codes a program that eventually halts when started on input x (all other registers set to zero).*

(2) **Decider** *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 **Decider** *halts with output 2 in $R1$.*

Proof. Assume that **Decider** is such a RAM program, and let Q be the following program with a single input:

$$\text{Program } Q \text{ (code } q) \begin{cases} & R2 \leftarrow & R1 \\ & \textbf{Decider} & \\ N1 & \texttt{continue} & \\ & R1 \ \texttt{jmp} & N1a \\ & \texttt{continue} & \end{cases}$$

Let i be the code of some program P.

Key point: *the termination behavior of Q on input i is exactly the opposite of the termination behavior of* **Decider** *on input i and code i.*

(1) If **Decider** says that program P coded by i *halts* on input i, then $R1$ just after the `continue` in line $N1$ contains 1, and Q *loops forever*.

(2) If **Decider** says that program P coded by i *loops forever* on input i, then $R1$ just after `continue` in line $N1$ contains 0, and Q *halts*.

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 the program Q on input q in $R1$ (all other registers set to zero).

Just after execution of the assignment $R2 \leftarrow R1$, the program **Decider** is started with q in both $R1$ and $R2$.

Since **Decider** is supposed to halt for all inputs, it eventually halts with output 0 or 1 in $R1$.

If **Decider** halts with output 1 in $R1$, then Q goes into an infinite loop, while if **Decider** halts with output 0 in $R1$, then Q halts.

But then, because of the definition of **Decider**, we see that **Decider** 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, **Decider** 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 is 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 6.1. 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 is the ith one, and z is the code of the current contents of the registers*.

Lemma 6.2. *The functions* Nextline, Nextcont, *and* Comp, *are (primitive) recursive.*

We can now prove that every RAM computable function can be computed in such a way that only one while loop is needed (all the other loops involve a fixed number of iterations).

Such a function is partial recursive in the sense of Kleene.

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 P and y codes the contents of its registers, $\mathrm{End}(x, y)$ *is the number of steps for which the computation of P runs before halting, if it halts*.

If the program does not halt, then $\mathrm{End}(x, y)$ is undefined.

We define $\mathrm{End}(x, y)$ as

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

Since $\mathrm{Comp}(x, y, m) = \langle i, z\rangle$, we have

$$\Pi_1(\mathrm{Comp}(x, y, m)) = i,$$

where i is the number (index) of the instruction reached after running the program P coded by x with initial values of the registers coded by y for m steps.

Thus, P halts if i is the last instruction in P, namely $\mathrm{Ln}(x)$, iff

$$\Pi_1(\mathrm{Comp}(x, y, m)) = \mathrm{Ln}(x).$$

End is a partial recursive function; it can be computed by a RAM program involving only one while loop searching for the number of steps m.

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

If φ is the partial recursive function computed by the program P coded by x, then we have

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

This is because if $m = \mathrm{End}(x, \langle y, 0\rangle)$ is the number of steps after which the program P coded by x halts on input y, then

$$\mathrm{Comp}(x, \langle y, 0\rangle, m)) = \langle \mathrm{Ln}(x), z\rangle,$$

where z is the code of the register contents when the program stops.

Consequently

$$\begin{aligned}
z &= \Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, m))\\
z &= \Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle))).
\end{aligned}$$

The value of the register $R1$ is $\Pi_1(z)$, that is

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

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:

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

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 5.9, 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.

For example, the program Padd2 (adding 2 to an integer) appears as

$$P_{1\,018\,748\,519\,973\,070\,618}.$$

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 6.2. 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*.

For example, the function add2 (adding 2 to an integer) appears as

$$\varphi_{1\,018\,748\,519\,973\,070\,618}.$$

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.

It is important to observe that *different programs* P_x and P_y may compute the *same function*, that is, while $P_x \neq P_y$ for all $x \neq y$, it is possible that $\varphi_x = \varphi_y$.

In fact, it is *undecidable* whether $\varphi_x = \varphi_y$.

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

Lemma 6.3. *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 6.1 and Lemma 6.3, 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).

6.3 Undecidability and Reducibility

In Section 5.6 we defined the recursively enumerable languages and the recursive languages in terms of Turing machines.

In view of the equivalence of RAM-computability and Turing- computability it will be convenient to define such languages in terms of recursive or partial recursive functions.

Given a set $L \subseteq \mathbb{N}$ of more generally $L \subseteq \Sigma^*$, recall that the *characteristic function* C_L of L is defined by

$$C_L(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{if } x \notin L. \end{cases}$$

In other words, C_L decides membership in L.

We have the following equivalent definitions of the recursively enumerable languages and the recursive languages.

Definition 6.3. A set $L \subseteq \mathbb{N}$ (or $L \subseteq \Sigma^*$) is *recursive* (or *decidable*) if its characteristic function C_L is total recursive.

A set $L \subseteq \mathbb{N}$ (or $L \subseteq \Sigma^*$) is *recursively enumerable* (or *partially decidable*) if it is the domain of a partial recursive function.

A set $L \subseteq \mathbb{N}$ (or $L \subseteq \Sigma^*$) is *undecidable* iff L is *not* recursive.

Thus, a set L is recursively enumerable iff there is a partial recursive function $f\colon \mathbb{N} \to \mathbb{N}$ (or $f\colon \Sigma^* \to \Sigma^*$) such that

$$f(x) \quad \text{is defined} \quad \text{iff} \quad x \in L.$$

If we think of f as computed by a Turing machine, then this is equivalent to Definition 5.11.

The following important result is a special case of Lemma 7.9.

Lemma 6.4. *A set $L \subseteq \mathbb{N}$ (or $L \subseteq \Sigma^*$) is recursively enumerable if and only if either $L = \emptyset$ or L it the range of a total recursive function f; that is, $L = f(\mathbb{N})$ (or $L = f(\Sigma^*)$).*

Intuitively, the recursive function f is a method for effectively listing all (and only) elements in L.

A closer look at the proof of the undecidability of the halting problem (Theorem 6.1) shows that the set of codes of RAM programs that halt on their own code as input

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

is *not* recursive.

However, since K is the domain of the partial recursive function $f(x) = \varphi_{univ}(x, x)$, it is recursively enumerable.

Therefore, *the set K is a set that is recursively enumerable but not recursive*.

The set K is partially decidable but undecidable.

Lemma 6.5. *A set L is recursive iff both L and $\overline{L}$ are recursively enumerable.*

For a proof, see the proof of Lemma 7.8.

From the above, we conclude that

$$\overline{K} = \{x \in \mathbb{N} \mid \varphi_x(x) \text{ is undefined}\}$$

is *not* recursively enumerable.

The undecidability of the halting problem (Theorem 6.1) also shows that the set

$$K_0 = \{\langle x, y\rangle \in \mathbb{N} \mid \varphi_x(y) \text{ is defined}\}$$

is *not* recursive. This set is an encoding of the halting problem.

However, since K_0 is the domain of the partial recursive function $f(z) = \varphi_{univ}(\Pi_1(z), \Pi_2(z))$, it is recursively enumerable.

The set K_0 is another set that is recursively enumerable but not recursive.

The set K_0 is partially decidable but undecidable.

By Lemma 6.5, the set $\overline{K_0}$ is not recursively enumerable.

Even more surprising, the set

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

is *not* recursively enumerable. We will prove this later.

This shows that the notion of a total recursive function is a very elusive notion, from a computable point of view.

We can't even enumerate recursively the total recursive functions!

Lemma 6.6. *If f and g are any two (total) recursive functions, then their composition $g \circ f$ is also (total) recursive. Similarly, if f and g are any two partial recursive functions, then their composition $g \circ f$ is also partial recursive.*

The above is easily proved using RAM programs.

Consider the set

$$H_0 = \{x \in \mathbb{N} \mid \varphi_x(0) \text{ is defined}\},$$

the set of codes of RAM programs that halt on input 0.

We claim that H_0 is not recursive, but how do we prove this?

We use a technique known as *reducibility*.

We construct a *(total) recursive* function f such that:

Given an integer i, the code of the RAM program P_i, the number $f(i)$ is the code of the program $P_{f(i)}$ obtained from P_i by adding instructions before P_i to initialize register $R1$ with the value i.

This new program $P_{f(i)}$ *ignores the initial value of its input and replaces it by i. After that, it simulates P_i on input i*.

Thus, observe that *P_i halts on input i iff $P_{f(i)}$ halts on input 0 (since $P_{f(i)}$ ignores its input and then simulates P_i on input i)*.

This fact can be stated as

$$i \in K \quad \text{iff} \quad f(i) \in H_0.$$

Therefore, if we had an algorithm to decide recursively membership in H_0, namely if C_{H_0} was recursive, then we would have an algorithm to decide recursively membership in K, since $C_K = C_{H_0} \circ f$ is also recursive as the composition of two recursive functions.

However K is not recursive, so H_0 is not recursive either.

The above is an instance of reducibility.

Definition 6.4. Let A and B be subsets of $\mathbb{N}$ (or Σ^*). We say that the set A is *many-one reducible* to the set B if there is a *total recursive* function $f\colon \mathbb{N} \to \mathbb{N}$ (or $f\colon \Sigma^* \to \Sigma^*$) such that

$$x \in A \quad \text{iff} \quad f(x) \in B \quad \text{for all } x \in \mathbb{N}.$$

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

Intuitively, deciding membership in B is as hard as deciding membership in A.

This is because any method for deciding membership in B can be converted to a method for deciding membership in A by first applying f to the number (or string) to be tested.

Here is another example of the use of reducibility to show that a set is not recursive.

Let us prove that

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

is not recursive by providing a reduction from H_0 to TOTAL.

We construct a *(total) recursive* function f such that:

Given an integer i, the code of the RAM program P_i, the number $f(i)$ is the code of the program $P_{f(i)}$ obtained from P_i by adding instructions before P_i to initialize register $R1$ with the value 0.

The program $P_{f(i)}$ *ignores the initial value of its input and replaces it by* 0*. After that, it simulates* P_i *on input* 0.

Now, observe that *P_i halts for input 0 iff $P_{f(i)}$ halts for all inputs (since $P_{f(i)}$ ignores its input and then simulates P_i on input 0)*.

This fact can be stated as

$$i \in H_0 \quad \text{iff} \quad f(i) \in \text{TOTAL}.$$

Therefore, if we had an algorithm to decide recursively membership in TOTAL, namely if C_{TOTAL} was recursive, then we would have an algorithm to decide recursively membership in H_0, since $C_{H_0} = C_{\text{TOTAL}} \circ f$ is also recursive as the composition of two recursive functions.

However H_0 is not recursive, so TOTAL is not recursive either.

We have the following general result.

Lemma 6.7. *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.

In most cases, we use (4) and (6).

A remarkable (and devastating) result of Rice shows that all nontrivial sets of partial recursive functions are not recursive.

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 $P_C \neq \mathbb{N}$.

Theorem 6.8. *(Rice's Theorem) 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.

For proof of Theorem 6.8, see the proof of Theorem 7.6. The idea is construct a reduction from K to P_C, where C is any nontrivial set of partial recursive functions.

Rice's Theorem shows that all nontrivial properties of the input/output behavior of programs are undecidable!

The scenario to apply Rice's Theorem to a class C of partial functions is to show that *some partial recursive function belongs to C* (C is not empty), *and that some partial recursive function does not belong to C* (C is not all the partial recursive functions). This demonstrates that C is nontrivial.

For example, in (a) of the next lemma, we need to exhibit a constant (partial) recursive function, such as $zero(n) = 0$, and a nonconstant (partial) recursive function, such as the identity function (or $succ(n) = n + 1$).

In particular, the following properties are undecidable.

Lemma 6.9. *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.

We conclude with the following crushing result which shows that TOTAL is not only undecidable, but not even partially decidable.

Lemma 6.10. *The set*

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

is not recursively enumerable.

Proof. 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. ☐

6.4 Kleene's T-Predicate

The object of this 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.

In Section 6.2, 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 (not the line number) 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. Thus, $i_{n-1} = \mathrm{Ln}(x)$ and i_n is irrelevant. Writing the definition of T is a little simpler if we let $i_n = \mathrm{Ln}(x) + 1$.

Also, y_0 codes the initial contents of the registers, that is, $y_0 = \langle y, 0 \rangle$, for some input y. We let $\mathrm{Ln}(z) = \Pi_1(z)$.

Definition 6.5. The *T-predicate* is the (primitive) recursive predicate defined as follows:

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

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:

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

Using the T-predicate, we get the so-called Kleene normal form.

Theorem 6.11. *(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 $i \geq 2$, Cumclr(i) is the code of the program which clears registers $R2, \ldots, Ri$.

To get Cumclr, we can use the function clr(i) such that 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 6.12. *There is a (primitive) recursive function c such that*

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