Chapter 2

Universal RAM Programs and Undecidability of the Halting Problem

The goal of this chapter is to prove three of the main results of computability theory:

(1) The undecidability of the halting problem for RAM programs (and Turing machines).

(2) The existence of universal RAM programs.

(3) The existence of the Kleene T-predicate.

All three require the ability to code a RAM program as a natural number.

Gödel pioneered the technique of encoding objects such as proofs as natural numbers in his famous paper on the (first) incompleteness theorem (1931).

One of the technical issues is to *code (pack) a tuple of natural numbers as a single natural number, so that the numbers being packed can be retrieved*.

Gödel designed a fancy function whose defintion does not involve recursion (Gödel's β function; see Kleene [**?**] or Shoenfield [**?**]).

For our purposes, a simpler function J due to Cantor packing two natural numbers m and n as a single natural number $J(m, n)$ suffices.

Another technical issue is the fact it is possible to reduce most of computability theory to numerical functions $f\colon \mathbb{N}^m \to \mathbb{N}$, and even to functions $f\colon \mathbb{N} \to \mathbb{N}$.

Indeed, there are primitive recursive coding and decoding functions $D_k\colon \Sigma^* \to \mathbb{N}$ and $C_k\colon \mathbb{N} \to \Sigma^*$ such that $C_k \circ D_k = \mathrm{id}_{\Sigma^*}$, where $\Sigma = \{a_1, \ldots, a_k\}$.

It is simpler to code programs (or Turing machines) taking natural numbers as input.

2.1 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:

$$\begin{tikzcd}
y \\
4 & 10 \arrow[dr] \\
3 & 6 \arrow[dr] & 11 \arrow[dr] \\
2 & 3 \arrow[dr] & 7 \arrow[dr] & 12 \arrow[dr] \\
1 & 1 \arrow[dr] & 4 \arrow[dr] & 8 \arrow[dr] & 13 \arrow[dr] \\
0 & 0 & 2 & 5 & 9 & 14 \\
 & 0 & 1 & 2 & 3 & 4 & x
\end{tikzcd}$$

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

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

Definition 2.1. The *pairing function* $J\colon \mathbb{N}^2 \to \mathbb{N}$ is defined by

$$J(x,y) = ((x+y)^2+3x+y)/2 \quad \text{for all } x,y \in \mathbb{N}.$$

The functions $K\colon \mathbb{N} \to \mathbb{N}$ and $L\colon \mathbb{N} \to \mathbb{N}$ are 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 $a,b \in \mathbb{N}$.

The functions J, K, L are called *Cantor's pairing functions*.

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 $J(x, y) < J(x, y')$.

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

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

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

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.

The notation $\langle x, y\rangle$ is “intentionally ambiguous,” in the sense that it can be interpreted as the actual ordered pair consisting of the two numbers x and y, or as *the number* $\langle x, y\rangle = J(x, y)$ that encodes the pair consisting of the two numbers x and y.

The context should make it clear which interpretation is intended. In this chapter and the next, it is the number (code) interpretation.

We can define bijections between $\mathbb{N}^n$ and $\mathbb{N}$ by induction for all $n \geq 1$.

Definition 2.2. The function $\langle -, \ldots, - \rangle_n \colon \mathbb{N}^n \to \mathbb{N}$ called an *extended pairing function* is defined as follows. We let

$$\begin{aligned} \langle z \rangle_1 &= z \\ \langle x_1, x_2 \rangle_2 &= \langle x_1, x_2 \rangle, \end{aligned}$$

and

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

for all $z, x_2, \ldots, x_{n+1} \in \mathbb{N}$.

Again we stress that $\langle x_1, \ldots, x_n \rangle_n$ is a *natural number*.

For example.

$$\begin{aligned}
\langle x_1, x_2, x_3\rangle_3 &= \langle x_1, \langle x_2, x_3\rangle\rangle_2 \\
&= \langle x_1, \langle x_2, x_3\rangle\rangle \\
\langle x_1, x_2, x_3, x_4\rangle_4 &= \langle x_1, x_2, \langle x_3, x_4\rangle\rangle_3 \\
&= \langle x_1, \langle x_2, \langle x_3, x_4\rangle\rangle\rangle \\
\langle x_1, x_2, x_3, x_4, x_5\rangle_5 &= \langle x_1, x_2, x_3, \langle x_4, x_5\rangle\rangle_4 \\
&= \langle x_1, \langle x_2, \langle x_3, \langle x_4, x_5\rangle\rangle\rangle\rangle.
\end{aligned}$$

It can be shown by induction on n that

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

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 an n-tuple, and $\Pi(i, n, z)$ as the i-th component of that n-tuple, but if z, n and i do not fit this interpretation, the function must be still be defined and we give it a "crazy" value by default using some simple primitive recursive clauses.

Definition 2.3. The *uniform projection function* $\Pi\colon \mathbb{N}^3 \to \mathbb{N}$ 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 the results of Section 1.8, this is a legitimate primitive recursive definition.

When $i = 0$ or $i > n + 1$, we get "bogus" values.

Remark: One might argue that it would have been preferable to order the arguments of Π as (n, i, z) rather than (i, n, z).

We use the order (i, n, z) in conformity with Machtey and Young [**?**].

Some basic properties of Π are given as exercises; see the notes.

As a first application, we observe that we need only consider partial computable functions of a single argument.

Indeed, let $\varphi\colon \mathbb{N}^n \to \mathbb{N}$ be a partial computable 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 computable 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 computable functions over Σ^* can be recoded as partial computable functions over $\{a_1\}^*$. For details, see the notes.

Since $\{a_1\}^*$ is isomorphic to $\mathbb{N}$, this shows that we can restrict out attention to functions defined over $\mathbb{N}$.

2.2 Coding of RAM Programs; The Halting Problem

In this section we present a specific encoding of RAM programs which allows us *to treat programs as integers*.

This encoding will allow us to prove one of the most important results of computabilty theory first proven by Turing for Turing machines (1936-1937), the *undecidability of the halting problem* for RAM programs (and Turing machines).

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).

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.

Definition 2.4. Instructions of a RAM program (operating on $\mathbb{N}$) are coded as follows:

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

The code of an instruction I is denoted as $\#I$.

To simplify the notation, we introduce the following decoding primitive recursive functions Typ, LNum, Reg, and Jmp, defined as follows:

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

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 $\Rightarrow$ (*implies*), defined such that

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

Definition 2.5.

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

The predicate $\mathrm{INST}(x)$ says that if x is the code of an instruction, say $x = \langle c, i, j, k \rangle$, then $1 \leq c \leq 5$, $j \geq 1$, if $c \leq 3$, then $k = 0$, and if $c = 0$ then we also have $j = 1$.

Definition 2.6. 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 Property $(*)$ in Section 2.1 that

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

Also recall that

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

Example 2.1. Consider the following program Padd2 computing the function add2: $\mathbb{N} \to \mathbb{N}$ given by

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

```
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:

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

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.

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`.

Definition 2.7. The primitive recursive predicate PROG(x) is given by

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

Note that we have used Proposition 1.7 which states that if f is a primitive recursive function and if 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 2.1. *(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:

```
                         $R2$ ←      $R1$
                              Decider
Program $Q$ (code $q$)  $N1$       continue
                         $R1$ jmp    $N1a$
                              continue
```

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. $\square$

The argument used in the proof of 2.1 is quite similar in spirit to “Russell’s Paradox.”

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.

2.3 Universal RAM Programs

To describe the computation of a RAM program, we need to code not only RAM programs but also the contents of the registers.

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:

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

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.

There are a lot of tedious technical details that the reader should skip upon first reading. However, to be rigorous, we must spell out all these details.

The most important function to define is Comp, and this function requires the auxiliary functions Nextline and Nextcont.

For the sake of simplicity we only define Comp; full details are given in the notes.

Definition 2.8. Let x code a program and let i be such that $1 \leq i \leq \mathrm{Ln}(x)$. The function Comp is defined such that

$\mathrm{Comp}(x, y, m) = \langle i, z \rangle$, 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 2.2. *The functions* Nextline*,* Nextcont*, and* Comp*, are (primitive) recursive.*

We can now reprove that every RAM computable function is partial computable. Indeed, assume that x codes a program P.

We would like to 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.

The following definition. works (see the notes).

Definition 2.9. The predicate $\mathrm{End}(x, y)$ is defined by

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

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

The function involved in the minimization is *primitive recursive*. However, in general, End is not a total function.

If φ is the partial computable function computed by the program P coded by x, then we claim that

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

The above fact is worth recording as the following proposition which is a variant of a result known as the *Kleene normal form*

Proposition 2.3. *(Kleene normal form for RAM programs) If φ is the partial computable function computed by the program P coded by x, then we have*

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

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

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

It will be convenient to denote the function φ computed by the RAM program coded by x as φ_x.

We can also exhibit a partial computable function which enumerates all the unary partial computable 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 computable 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 computable 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 Proposition 1.9, the partial function φ_{univ} is not computable (recursive).[1]

Indeed, being an enumerating function for the partial computable functions, it is an enumerating function for the total computable functions, and thus, it cannot be computable.

Being a partial function saves us from a contradiction.

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

Proposition 2.4. *(Universal RAM program) For the indexing of RAM programs defined earlier, there is a universal partial computable function φ_{univ} such that, for all $x, y \in \mathbb{N}$, if φ_x is the partial computable 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.

[1]The term *recursive function* is now considered old-fashion. Many researchers have switched to the term *computable function*.

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 2.1 and Proposition 2.4, 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 existence of the function φ_{univ} leads us to the notion of an *indexing* of the RAM programs.

2.4 Indexing of 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 computable 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 computable functions.

Definition 2.10. 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 computable 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 computable 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 $\varphi_x = \varphi_y$.

2.5 Undecidability and Reducibility

In Section 1.5 we defined the listable (computably enumerable) languages and the computable 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 computable or partial computable 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 listable (computably enumerable) languages and the computable languages.

Definition 2.11. A set $L \subseteq \mathbb{N}$ (or $L \subseteq \Sigma^*$) is *computable* (or *recursive*) (or *decidable*) if its characteristic function C_L is total computable.

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

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

Thus, a set L is listable (computably enumerable) iff there is a partial computable 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 1.12.

The following important result is a special case of Lemma proven in the notes.

Lemma 2.5. *A set $L \subseteq \mathbb{N}$ (or $L \subseteq \Sigma^*$) is listable (computably enumerable) if and only if either $L = \emptyset$ or L is the range of a total computable function f; that is, $L = f(\mathbb{N})$ (or $L = f(\Sigma^*)$).*

Intuitively, the computable 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 2.1) 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* computable (not recursive).

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

Therefore, *the set K is a set that is listable but not computable (not recursive)*.

Proposition 2.6. *A set L is computable (recursive) iff both L and $\overline{L}$ are listable (computably enumerable).*

For a proof, see the notes.

From the above, we conclude that

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

is *not* listable.

The undecidability of the halting problem (Theorem 2.1) also shows that the set

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

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

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

The set K_0 is another set that is listable but not computable (not recursive).

By Proposition 2.6, the set $\overline{K_0}$ is not listable.

Even more surprising, the set

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

is *not* listable. We will prove this later.

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

We can't even enumerate computably the total computable functions!

Due to their importance let us record K, K_0 and TOTAL in the following definition:

Definition 2.12. The sets of natural numbers K, K_0 and TOTAL are defined as follows:

$$\begin{aligned} K &= \{x \in \mathbb{N} \mid \varphi_x(x) \text{ is defined}\} \\ K_0 &= \{\langle x, y\rangle \in \mathbb{N} \mid \varphi_x(y) \text{ is defined}\} \\ \mathrm{TOTAL} &= \{x \mid \varphi_x \text{ is a total function}\}. \end{aligned}$$

Both K and K_0 are listable but not computable, and TOTAL is not even listable.

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 computable, but how do we prove this?

We use a technique known as *reducibility*.

We construct a *(total) computable 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 computably membership in H_0, namely if C_{H_0} was computable, then we would have an algorithm to decide computably membership in K, since $C_K = C_{H_0} \circ f$ is also computable as the composition of two computable functions.

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

The above is an instance of reducibility.

Definition 2.13. 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 computable* function (*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 $A \leq B$ (or more precisely $A \leq_m B$), 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 computable (not recursive).

Let us prove that

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

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

We construct a *(total) computable* 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 computably membership in TOTAL, namely if C_{TOTAL} was computable, then we would have an algorithm to decide computably membership in H_0, since $C_{H_0} = C_{\text{TOTAL}} \circ f$ is also computable as the composition of two computable functions.

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

We have the following general result.

Proposition 2.7. *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 c.e. (r.e.), then A is c.e. (r.e.).

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

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

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

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

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

Let C be any set of partial computable 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 computable functions. For example

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

We say that C is *nontrivial* if C is neither empty nor the set of all partial computable functions.

Equivalently C is nontrivial iff $P_C \neq \emptyset$ and $P_C \neq \mathbb{N}$.

Theorem 2.8. *(Rice's Theorem) For any set C of partial computable functions, the set*

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

is not computable (not recursive) unless C is trivial.

For proof of Theorem 2.8, see the notes.

The idea is construct a reduction from K to P_C, where C is any nontrivial set of partial computable functions.

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

It is important to understand that Rice's theorem says that *the set P_C of indices of all partial computable functions equal to some function in a given set C of partial computable functions is not computable* if C is nontrivial, *not that the set C is not computable* if C is nontrivial.

The second statement does not make any sense because our machinery only applies to sets of natural numbers (or sets of strings).

For example, the set $C = \{\varphi_{i_0}\}$ consisting of a single partial computable function is nontrivial, and being finite, under the second wrong interpretation it would be computable.

But we need to consider the set

$$P_C = \{n \in \mathbb{N} \mid \varphi_n = \varphi_{i_0}\}$$

of indices of all partial computable functions φ_n that are equal to φ_{i_0}, and by Rice's theorem, this set is not computable.

In other words, it is undecidable whether an arbitrary partial computable function is equal to some fixed partial computable function.

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

This demonstrates that C is nontrivial.

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

In particular, the following properties are undecidable.

Proposition 2.9. *The following properties of partial computable functions are undecidable.*

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

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

(c) Two partial computable functions φ_x and φ_y are identical. More precisely, the set $\{\langle x, y\rangle \mid \varphi_x = \varphi_y\}$ is not computable.

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

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

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

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

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

Proposition 2.10. *The set*

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

is not listable (not recursively enumerable).

Proof. If TOTAL was listable, then there would be a total computable 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 computable. 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 listable. $\square$

2.6 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 2.2, 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.

It can be shown that the predicate T is (primitive) recursive; see the notes.

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 2.11. *(Kleene Normal Form) Using the indexing of the partial computable 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

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

2.7 A Non-Computable Function; Busy Beavers

Total functions that are not computable must grow very fast and thus are very complicated.

Yet, in 1962, Radó published a paper in which he defined two functions Σ and S (involving computations of Turing machines) that are total and not computable.

Consider Turing machines with a tape alphabet $\Gamma = \{1, B\}$ with two symbols (B being the blank).

We also assume that these Turing machines have a special final state q_F, which is a blocking state (there are no transitions from q_F).

We do not count this state when counting the number of states of such Turing machines.

The game is to run such Turing machines with a fixed number of states n starting on a blank tape, *with the goal of producing the maximum number of (not necessarily consecutive) ones* (1).

Definition 2.14. The function Σ (defined on the positive natural numbers) is defined as the maximum number $\Sigma(n)$ of (not necessarily consecutive) 1's written on the tape after a Turing machine with $n \geq 1$ states started on the blank tape halts.

The function S is defined as the maximum number $S(n)$ of moves that can be made by a Turing machine of the above type with n states before it halts, started on the blank tape.

A Turing machine with n states that writes the maximum number $\Sigma(n)$ of 1's when started on the blank tape is called a *busy beaver*.

Busy beavers are hard to find, even for small n.

First, it can be shown that the number of distinct Turing machines of the above kind with n states is $(4(n+1))^{2n}$.

Second, since it is undecidable whether a Turing machine halts on a given input, it is hard to tell which machines loop or halt after a very long time.

Here is a summary of what is known for $1 \leq n \leq 6$. Observe that the exact value of $\Sigma(5), \Sigma(6), S(5)$ and $S(6)$ is unknown.

n	$\Sigma(n)$	$S(n)$
1	1	1
2	4	6
3	6	21
4	13	107
5	≥ 4098	$\geq 47,176,870$
6	$\geq 95,524,079$	$\geq 8,690,333,381,690,951$
6	$\geq 3.515 \times 10^{18267}$	$\geq 7.412 \times 10^{36534}$

The first entry in the table for $n = 6$ corresponds to a machine due to Heiner Marxen (1999). T

his record was surpassed by Pavel Kropitz in 2010, which corresponds to the second entry for $n = 6$.

The machines achieving the record in 2017 for $n = 4, 5$ are shown below, where the blank is denoted Δ instead of B, and where the special halting states is denoted H:

4-state busy beaver:

	A	B	C	D
Δ	$(1, R, B)$	$(1, L, A)$	$(1, R, H)$	$(1, R, D)$
1	$(1, L, B)$	(Δ, L, C)	$(1, L, D)$	(Δ, R, A)

The above machine output 13 ones in 107 steps. In fact, the output is

$$\Delta\,\Delta\,1\,\Delta\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,\Delta\,\Delta.$$

5-state best contender:

	A	B	C	D	E
Δ	$(1, R, B)$	$(1, R, C)$	$(1, R, D)$	$(1, L, A)$	$(1, R, H)$
1	$(1, L, C)$	$(1, R, B)$	(Δ, L, E)	$(1, L, D)$	(Δ, L, A)

The above machine output 4098 ones in $47, 176, 870$ steps.

The reason why it is so hard to compute Σ and S is that they are not computable!

Theorem 2.12. *The functions Σ and S are total functions that are not computable (not recursive).*

The "zoo" of computable and non-computable functions is illustrated in Figure 2.1.

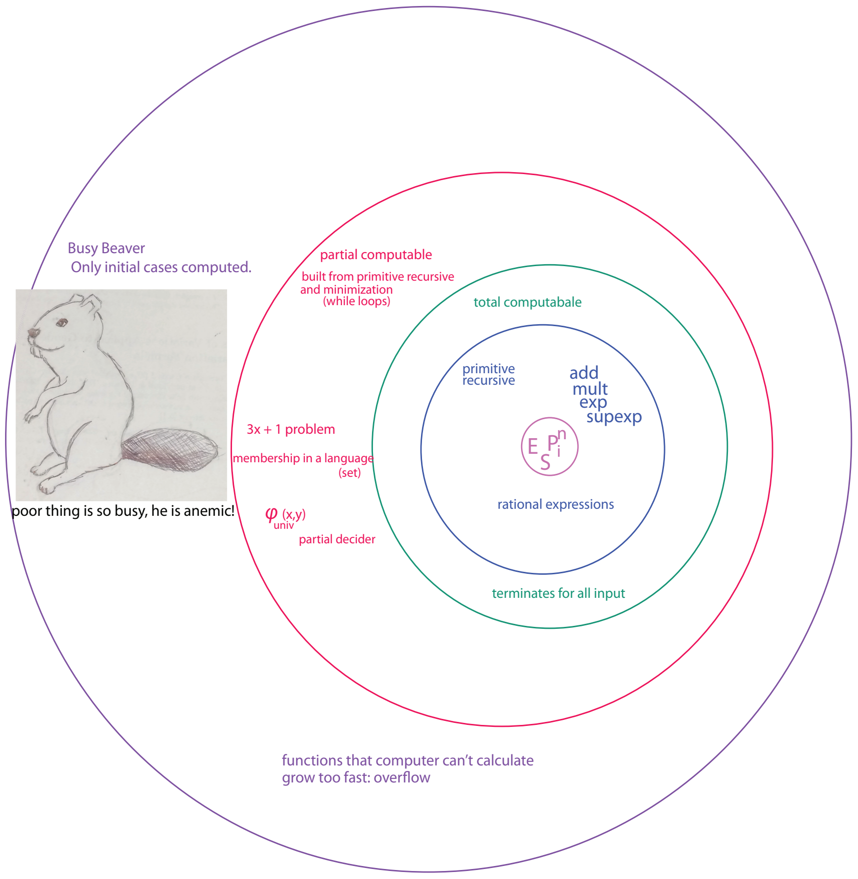

Figure 2.1: Computability Classification of Functions.