Chapter 5

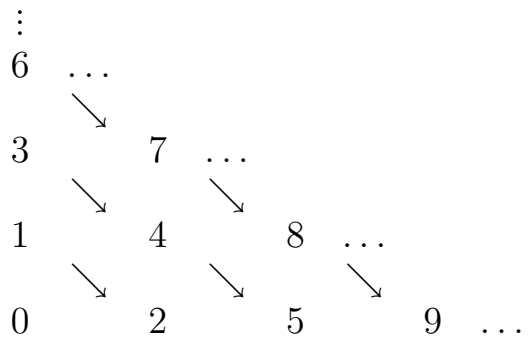
Universal RAM Programs and Undecidability of the Halting Problem

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*,
$J: \mathbb{N}^2 \to \mathbb{N}$.

The function J has the graph partially showed below:



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 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,

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

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 $\mathbb{N}$.

5.2 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).

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.

We can define the primitive recursive predicate INST, such that 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:

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

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.

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

The primitive recursive predicate PROG is defined such that $\mathrm{PROG}(x)$ 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

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

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 5.2.1 *(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 5.2.2 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 5.2.3 *The functions* Nextline*,* Nextcont*, and* Comp*, are primitive recursive.*

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 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:

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

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 4.8.2, 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 5.2.4 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 5.2.5 *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 5.2.1 and Lemma 5.2.5, 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.

5.3 Kleene's T-Predicate

In Section 5.2, 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 5.3.1 The *T-predicate* is the primitive recursive predicate defined as follows:

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

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:

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

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

Theorem 5.3.2 *(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 $i \geq 2$, 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 5.3.3 *There is a primitive recursive function c such that*

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