CHAPTER 2

Recursion Theory

In this chapter we develop the basics of recursive function theory, or as it is more generally known, computability theory. Its history goes back to the seminal works of Turing, Kleene and others in the 1930's.

A computable function is one defined by a program whose operational semantics tell an idealized computer what to do to its storage locations as it proceeds deterministically from input to output, without any prior restrictions on storage space or computation time. We shall be concerned with various program-styles and the relationships between them, but the emphasis throughout will be on one underlying data-type, namely the natural numbers, since it is there that the most basic foundational connections between proof theory and computation are to be seen in their clearest light.

The two best-known models of machine computation are the Turing Machine and the (Unlimited) Register Machine of Shepherdson and Sturgis (1963). We base our development on the latter since it affords the quickest route to the results we want to establish.

2.1. Register machines

2.1.1. Programs. A *register machine* stores natural numbers in registers denoted u, v, w, x, y, z possibly with subscripts, and it responds step by step to a *program* consisting of an ordered list of basic instructions:

$$\begin{array}{l} I_0 \\ I_1 \\ \vdots \\ I_{k-1} \end{array}$$

Each instruction has one of the following three forms whose meanings are obvious:

Zero: $x := 0$,
Succ: $x := x + 1$,
Jump: $[\textbf{if } x = y \textbf{ then } I_n \textbf{ else } I_m]$.

The instructions are obeyed in order starting with I_0 except when a conditional jump instruction is encountered, in which case the next instruction

will be either I_n or I_m according as the numerical contents of registers x and y are equal or not at that stage. The computation *terminates* when it runs out of instructions, that is when the next instruction called for is I_k. Thus if a program of length k contains a jump instruction as above then it must satisfy the condition $n, m \leq k$ and I_k means "halt". Notice of course that some programs do not terminate, for example the following one-liner:

$$[\textbf{if } x = x \textbf{ then } I_0 \textbf{ else } I_1]$$

2.1.2. Program constructs. We develop some shorthand for building up standard sorts of programs.

Transfer. "$x := y$" is the program

$$\begin{array}{l} x := 0 \\ [\textbf{if } x = y \textbf{ then } I_4 \textbf{ else } I_2] \\ x := x + 1 \\ [\textbf{if } x = x \textbf{ then } I_1 \textbf{ else } I_1], \end{array}$$

which copies the contents of register y into register x.

Predecessor. The program "$x := y \dot{-} 1$" copies the modified predecessor of y into x, and simultaneously copies y into z:

$$\begin{array}{l} x := 0 \\ z := 0 \\ [\textbf{if } x = y \textbf{ then } I_8 \textbf{ else } I_3] \\ z := z + 1 \\ [\textbf{if } z = y \textbf{ then } I_8 \textbf{ else } I_5] \\ z := z + 1 \\ x := x + 1 \\ [\textbf{if } z = y \textbf{ then } I_8 \textbf{ else } I_5]. \end{array}$$

Composition. "P **;** Q" is the program obtained by concatenating program P with program Q. However in order to ensure that jump instructions in Q of the form "$[\textbf{if } x = y \textbf{ then } I_n \textbf{ else } I_m]$" still operate properly within Q they need to be re-numbered by changing the addresses n, m to $k+n, k+m$ respectively where k is the length of program P. Thus the effect of this program is to do P until it halts (if ever) and then do Q.

Conditional. "**if** $x = y$ **then** P **else** Q **fi**" is the program

$$\begin{array}{l} [\textbf{if } x = y \textbf{ then } I_1 \textbf{ else } I_{k+2}] \\ \vdots P \\ [\textbf{if } x = x \textbf{ then } I_{k+2+l} \textbf{ else } I_2] \\ \vdots Q \end{array}$$

where k, l are the lengths of the programs P, Q respectively, and again their jump instructions must be appropriately renumbered by adding 1 to the addresses in P and $k+2$ to the addresses in Q. Clearly if $x = y$ then program P is obeyed and the next jump instruction automatically bypasses Q and halts. If $x \neq y$ then program Q is performed.

For Loop. "**for** $i = 1 \dots x$ **do** P **od**" is the program

$$\begin{array}{l} i := 0 \\ [\textbf{if } x = i \textbf{ then } I_{k+4} \textbf{ else } I_2] \\ i := i + 1 \\ \vdots P \\ [\textbf{if } x = i \textbf{ then } I_{k+4} \textbf{ else } I_2] \end{array}$$

where again, k is the length of program P and the jump instructions in P must be appropriately re-addressed by adding 3. The intention of this new program is that it should iterate the program P x times (do nothing if $x = 0$). This requires the restriction that the register x and the "local" counting-register i are not re-assigned new values inside P.

While Loop. "**while** $x \neq 0$ **do** P **od**" is the program

$$\begin{array}{l} y := 0 \\ [\textbf{if } x = y \textbf{ then } I_{k+3} \textbf{ else } I_2] \\ \vdots P \\ [\textbf{if } x = y \textbf{ then } I_{k+3} \textbf{ else } I_2] \end{array}$$

where again, k is the length of program P and the jump instructions in P must be re-addressed by adding 2. This program keeps on doing P until (if ever) the register x becomes 0; it requires the restriction that the auxiliary register y is not re-assigned new values inside P.

2.1.3. Register machine computable functions. A register machine program P may have certain distinguished "input registers" and "output registers". It may also use other "working registers" for scratchwork and these will initially be set to zero. We write $P(x_1, \dots, x_k; y)$ to signify that program P has input registers $x_1, \dots, x_k$ and one output register y, which are distinct.

DEFINITION. The program $P(x_1, \dots, x_k; y)$ is said to *compute* the k-ary partial function $\varphi\colon \mathbb{N}^k \to \mathbb{N}$ if, starting with any numerical values $n_1, \dots, n_k$ in the input registers, the program terminates with the number m in the output register if and only if $\varphi(n_1, \dots, n_k)$ is defined with value m. In this case, the input registers hold their original values.

A function is *register machine computable* if there is some program which computes it.

Here are some examples.

Addition. "Add$(x, y; z)$" is the program

$$z := x \ ; \ \textbf{for } i = 1, \ldots, y \ \textbf{do } z := z + 1 \ \textbf{od}$$

which adds the contents of registers x and y into register z.

Subtraction. "Subt$(x, y; z)$" is the program

$$z := x \ ; \ \textbf{for } i = 1, \ldots, y \ \textbf{do } w := z \dot{-} 1 \ ; \ z := w \ \textbf{od}$$

which computes the modified subtraction function $x \dot{-} y$.

Bounded Sum. If $P(x_1, \ldots, x_k, w; y)$ computes the $k+1$-ary function φ then the program $Q(x_1, \ldots, x_k, z; x)$:

$x := 0$;
for $i = 1, \ldots, z$ **do** $w := i \dot{-} 1$; $P(\vec{x}, w; y)$; $v := x$; Add$(v, y; x)$ **od**

computes the function

$$\psi(x_1, \ldots, x_k, z) = \sum_{w<z} \varphi(x_1, \ldots, x_k, w)$$

which will be undefined if for some $w < z$, $\varphi(x_1, \ldots, x_k, w)$ is undefined.

Multiplication. Deleting "$w := i \dot{-} 1$; P" from the last example gives a program Mult$(z, y; x)$ which places the product of y and z into x.

Bounded Product. If in the bounded sum example, the instruction $x := x + 1$ is inserted immediately after $x := 0$, and if Add$(v, y; x)$ is replaced by Mult$(v, y; x)$, then the resulting program computes the function

$$\psi(x_1, \ldots, x_k, z) = \prod_{w<z} \varphi(x_1, \ldots, x_k, w).$$

Composition. If $P_j(x_1, \ldots, x_k; y_j)$ computes φ_j for each $j = 1, \ldots, n$ and if $P_0(y_1, \ldots, y_n; y_0)$ computes φ_0, then the program $Q(x_1, \ldots, x_k; y_0)$:

$$P_1(x_1, \ldots, x_k; y_1) \ ; \ \ldots \ ; \ P_n(x_1, \ldots, x_k; y_n) \ ; \ P_0(y_1, \ldots, y_n; y_0)$$

computes the function

$$\psi(x_1, \ldots, x_k) = \varphi_0(\varphi_1(x_1, \ldots, x_k), \ldots, \varphi_n(x_1, \ldots, x_k))$$

which will be undefined if any of the φ-subterms on the right hand side is undefined.

Unbounded Minimization. If $P(x_1, \ldots, x_k, y; z)$ computes φ then the program $Q(x_1, \ldots, x_k; z)$:

$y := 0$; $z := 0$; $z := z + 1$;
while $z \neq 0$ **do** $P(x_1, \ldots, x_k, y; z)$; $y := y + 1$ **od** ;
$z := y \dot{-} 1$

computes the function

$$\psi(x_1, \ldots, x_k) \ = \ \mu_y(\varphi(x_1, \ldots, x_k, y) = 0)$$

that is, the *least number* y such that $\varphi(x_1, \ldots, x_k, y')$ is defined for every $y' \leq y$ and $\varphi(x_1, \ldots, x_k, y) = 0$.

2.2. Elementary functions

2.2.1. Definition and simple properties. The *elementary functions* of Kalmár (1943) are those number-theoretic functions which can be defined explicitly by compositional terms built up from variables and the constants $0, 1$ by repeated applications of addition $+$, modified subtraction $\dot{-}$, bounded sums and bounded products.

By omitting bounded products, one obtains the *subelementary* functions.

The examples in the previous section show that all elementary functions are computable and totally defined. Multiplication and exponentiation are elementary since

$$m \cdot n = \textstyle\sum_{i<n} m \text{ and } m^n = \prod_{i<n} m$$

and hence by repeated composition, all exponential polynomials are elementary.

In addition the elementary functions are closed under

Definition by Cases.

$$f(\vec{n}\,) = \begin{cases} g_0(\vec{n}\,) & \text{if } h(\vec{n}\,) = 0 \\ g_1(\vec{n}\,) & \text{otherwise} \end{cases}$$

since f can be defined from g_0, g_1 and h by

$$f(\vec{n}\,) = g_0(\vec{n}\,) \cdot (1 \dot{-} h(\vec{n}\,)) + g_1(\vec{n}\,) \cdot (1 \dot{-} (1 \dot{-} h(\vec{n}\,))).$$

Bounded Minimization.

$$f(\vec{n}, m) = \mu_{k<m}(g(\vec{n}, k) = 0)$$

since f can be defined from g by

$$f(\vec{n}, m) = \sum_{i<m} \big(1 \dot{-} \sum_{k \leq i} (1 \dot{-} g(\vec{n}, k))\big).$$

Note: this definition gives value m if there is no $k < m$ such that $g(\vec{n}, k) = 0$. It shows that not only the elementary, but in fact the subelementary functions are closed under bounded minimization. Furthermore, we define $\mu_{k \leq m}(g(\vec{n}, k) = 0)$ as $\mu_{k<m+1}(g(\vec{n}, k) = 0)$.

Lemma.

(a) *For every elementary function* $f\colon \mathbb{N}^r \to \mathbb{N}$ *there is a number* k *such that for all* $\vec{n} = n_1, \ldots, n_r$,

$$f(\vec{n}\,) < 2_k(\max(\vec{n}\,))$$

where $2_0(m) := m$ *and* $2_{k+1}(m) := 2^{2_k(m)}$.

(b) *The function* $n \mapsto 2_n(1)$ *is not elementary.*

PROOF. (a). By induction on the build-up of the compositional term defining f. The result clearly holds if f is any one of the base functions:

$$f(\vec{n}\,) = 0 \text{ or } 1 \text{ or } n_i \text{ or } n_i + n_j \text{ or } n_i \dot{-} n_j.$$

If f is defined from g by application of bounded sum or product:

$$f(\vec{n}, m) = \sum_{i<m} g(\vec{n}, i) \text{ or } \prod_{i<m} g(\vec{n}, i)$$

where $g(\vec{n}, i) < 2_k(\max(\vec{n}, i))$ then we have

$$f(\vec{n}, m) \leq (2_k(\max(\vec{n}, m)))^m < 2_{k+2}(\max(\vec{n}, m))$$

using $n^n < 2^{2^n}$ (since $n^n < (2^{n-1})^n \leq 2^{2^n}$ for $n \geq 3$).

If f is defined from $g_0, g_1, \ldots, g_l$ by composition:

$$f(\vec{n}\,) = g_0(g_1(\vec{n}\,), \ldots, g_l(\vec{n}\,))$$

where for each $j \leq l$ we have $g_j(-) < 2_{k_j}(\max(-))$, then with $k = \max_j k_j$,

$$f(\vec{n}\,) < 2_k(2_k(\max(\vec{n}\,))) = 2_{2k}(\max(\vec{n}\,))$$

and this completes the first part.

(b). If $2_n(1)$ were an elementary function of n then by (a) there would be a positive k such that for all n,

$$2_n(1) < 2_k(n)$$

but then putting $n = 2_k(1)$ yields $2_{2_k(1)}(1) < 2_{2k}(1)$, a contradiction. □

2.2.2. Elementary relations. A relation R on $\mathbb{N}^k$ is said to be *elementary* if its characteristic function

$$c_R(\vec{n}\,) = \begin{cases} 1 & \text{if } R(\vec{n}\,) \\ 0 & \text{otherwise} \end{cases}$$

is elementary. In particular, the "equality" and "less than" relations are elementary since their characteristic functions can be defined as follows:

$$c_<(n, m) = 1 \dot{-} (1 \dot{-} (m \dot{-} n)), \quad c_=(n, m) = 1 \dot{-} (c_<(n, m) + c_<(m, n)).$$

Furthermore if R is elementary then so is the function

$$f(\vec{n}, m) = \mu_{k<m} R(\vec{n}, k)$$

since $R(\vec{n}, k)$ is equivalent to $1 \dot{-} c_R(\vec{n}, k) = 0$.

LEMMA. *The elementary relations are closed under applications of propositional connectives and bounded quantifiers.*